2005-08-23

Continuous Query Processing on Spatio-Temporal Data Streams

Rimma V. Nehme
Worcester Polytechnic Institute

Continuous Query Processing on Spatio-Temporal Data Streams

by

Rimma V. Nehme

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

June 2005

APPROVED:

Professor Elke A. Rundensteiner, Thesis Advisor

Professor Michael A. Gennert, Thesis Reader

Professor Michael A. Gennert, Head of Department

Abstract

This thesis addresses important challenges in the areas of streaming and spatio-temporal databases. It focuses on continuous querying of spatio-temporal environments characterized by (1) a large number of moving and stationary objects and queries; (2) need for near real-time results; (3) limited memory and cpu resources; and (4) different accuracy requirements.

The first part of the thesis studies the problem of performance vs. accuracy tradeoff using different location modelling techniques when processing continuous spatio-temporal range queries on moving objects. Two models for modeling the movement, namely: continuous and discrete models are described. This thesis introduces an accuracy comparison model to estimate the quality of the answers returned by each of the models. Experimental evaluations show the effectiveness of each model given certain characteristics of spatio-temporal environment (e.g., varying speed, location update frequency).

The second part of the thesis introduces SCUBA, a Scalable Cluster Based Algorithm for evaluating a large set of continuous queries over spatio-temporal data streams. Unlike the commonly used static grid indices, the key idea of SCUBA is to group moving objects and queries based on common *dynamic* properties (e.g., speed, destination, and road network location) at run-time into moving clusters. This results in improvement in performance which facilitate scalability. SCUBA exploits shared cluster-based execution consisting of two phases. In phase I, the evaluation of a set of spatio-temporal queries is abstracted as a spatial join between moving clusters for cluster-based filtering of true negatives. There after, in phase II, a fine-grained join process is executed for all pairs identified as potentially joinable by a positive cluster-join match in phase I. If the clusters don't satisfy the join predicate, the objects and queries that belong to those clusters can be savely discarded as being guaranteed to not join individually either. This provides processing cost savings.Another advantage of SCUBA is that moving cluster-driven load shedding is facilitated. A moving cluster (or its subset, called *nucleus*) approximates the locations of its members. As a consequence relatively accurate answers can be produced using solely the abstracted cluster location information in place of precise object-by-object matches, resulting in savings in memory and

improvement in processing time. A theoretical analysis of SCUBA is presented with respect to the memory requirements, number of join comparisons and I/O costs. Experimental evaluations on real datasets demonstrate that SCUBA achieves a substantial improvement when executing continuous queries on highly dense moving objects. The experiments are conducted in a real data streaming system (CAPE) developed at WPI on real datasets generated by the Network-Based Moving Objects Generator.

Acknowledgements

This thesis is by far the most significant scientific accomplishment in my life and it would be impossible without people who supported me and believed in me.

Most of all I would like to thank my research advisor, Professor Elke A. Rundensteiner, whose passion for research, teaching and scientific exploration inspired me in more ways than I can count. She taught me new ways to look at research and encouraged creativity, took the time to read and carefully comment on my papers, and gave me encouragement and practical advice when I needed it most. Her unlimited energy, dedication and enthusiasm was contagious. It was a great privilege to work with her on this thesis.

I have learned a great deal from my fellow graduate students and friends in DSRG and WPI. I particularly want to thank Bin Liu, Mariana Jbantova, Maged El-Sayed, Venky Raghavan, Luping Ding, Yali Zhu, Hong Su, Ling Wang, Song Wang, Brad Momberger and others for challenging, on-going exchanges of ideas. The members of the DSRG II, a.k.a *International House* all deserve a special mention for their practical advice, encouragement, laughs and occasional sanity checks.

I would also like to thank all of my professors at WPI for inspiring me to question things, think critically, and challenging me. In particular, I would like to thank Professor Michael Gennert, my Thesis Reader, and Professor George Heineman for regularly checking on my research progress.

My deepest thanks goes to my husband and my best friend Alfred (Freddy) for his unconditional love and support in everything. Thank you for bringing the best in me.

I want to thank my parents Nina and Vladimir Kaftanchikov for being my Mom and Dad and loving me more than anything in this world. A special thanks goes to my extended family scattered all over the world - Belarus, Lebanon, Kazakhstan, and Russia, for loving and believing in me.

Finally, a special gratitude goes to my host parents, John and Betsy Chapman for their love, support and for giving me a tremendeous opportunity to change my life.

Contents

List of Figures

List of Tables

Glossary

Centroid	Center mass of a cluster. The centroid of a set of multi-dimensional data points is the data point that is the mean of the values in each dimension.
CM	Competitive Learning Clustering Algorithm.
CSA	Continuous Scalable Trajectory-Based Algorithm.
DSA	Discrete Scalable Point-Based Algorithm.
False positives	Number of positive cases incorrectly detecting some condition.
False negatives	Number of negative cases incorrectly detecting some condition. (Number of cases that report something as not to happen when, in fact, it did.)
K-means	A common clustering algorithm for partitioning data into clusters. The algorithm returns k clusters, each of which contains data points that are more similar to points in the same cluster than to points in any other cluster.
LF	Leader-Follower Clustering Algorithm
Offline Algorithm	An algorithm which is given the entire sequence of inputs in advance. Also called non-incremental algorithm.
Online Algorithm	An algorithm that must process each input in turn, without detailed knowledge of future inputs. Also called incremental algorithm.
Update Probability	Variable in moving objects data generator describing the probability that an object sends its location update every time unit.
SCUBA	Scalable Cluster-Based Algorithm for evaluating continuous spatio-temporal queries on moving objects.

Chapter 1

Part I:

Introduction

1.1 Motivation

Every day we are witnessing continued improvements in wireless communication and geo-positioning. With the help of Global Positioning Systems (GPS), people can avoid congested freeways and find more efficient routes to their destinations, saving millions of dollars in gasoline and tons of air pollution. Travel aboard ships and aircraft is becoming safer in all weather conditions. Businesses with large amounts of transportation costs are able to manage their resources more efficiently, reducing consumer costs.

These developments spawned research in the recent years in the field of spatio-temporal databases and real-time streaming databases [17]. Many new applications utilizing the spatio-temporal aspects of data items begin to emerge. For example medical facilities can track staff and monitor patients for emergency response, and coordinate logistics in case of an emergency (e.g., navigate to the closest hospital or the nearest emergency department (ED) that has available capacity). Utilities and commercial service providers can track their service engineers and direct the closest service crew to solve a problem. Emergency response centers can help people navigate and evacuate faster

in case of disasters such as floods, earthquakes, tornadoes or terrorist attacks. Benefits and uses of location data are potentially boundless. For instance, scientific and educational applications can be built based on spatio-temporal data. A geologist, for example, driving through a terrain can use a hand-held device to view the area she sees with the naked eye, but with additional information superimposed, which may include seismographic charts, images of the terrain taken at another season, notes made by other geologists about each landmark in the viewable terrain [82].

Combining the functionality of locator technologies, global positioning systems (GPSs), wireless and cellular telephone technologies, and information technologies enables new environments where virtually all objects of interest can determine their locations. These technologies have the potential to improve the quality of life by adding location-awareness to virtually all objects of interest such as humans, cars, laptops, eyeglasses, canes, desktops, pets, wild animals, bicycles, and buildings. Figure 1.1 gives some real-world examples of spatio-temporal applications. These include location-aware services, traffic monitoring, asset tracking, personal safety, etc.

Unlike traditional databases, spatio-temporal databases typically deal with large number of objects that change their positions in space with time. The assumption is that eventually every object or a person would have a capability (technology-wise and business-wise) of reporting its location to some central server, and applications can be build utilizing this data.

Another distinguishing characteristics for such environments is that queries themselves can move. Thus the system needs to consider both the positions of the moving objects as well as queries [62]. This calls for new real-time spatio-temporal query processing algorithms that deal with large numbers of moving objects and large numbers of continuous spatio-temporal queries where near-real time response is a necessity.

A key point in spatio-temporal query processing is that any delay of the query response may result in an obsolete answer. For example, consider a query that asks about the moving objects that lie in a certain region. If the query answer is delayed, the answer may be outdated where objects are continuously changing their locations.

Secondly, spatio-temporal databases need to support a wide variety of continuous spatio-temporal

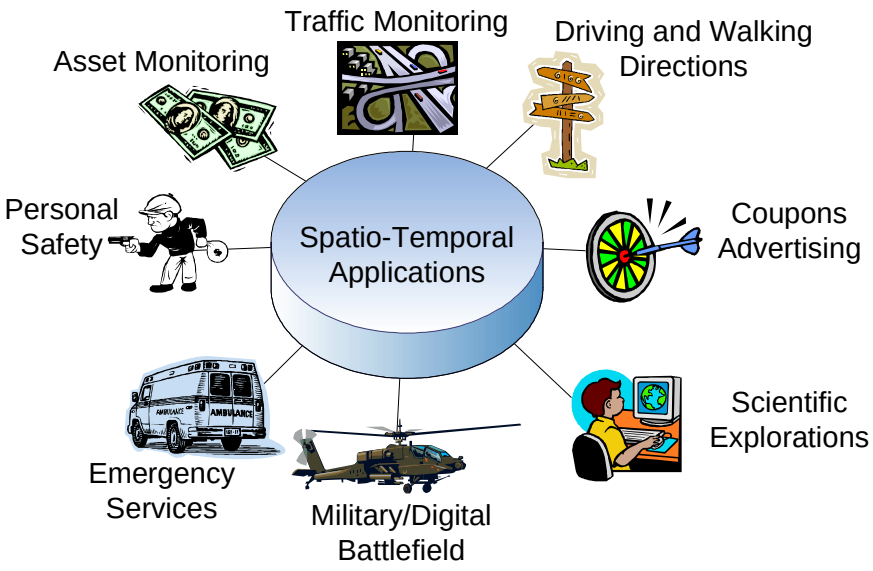


Figure 1.1: Spatio-temporal applications

queries. For example, a continuous spatio-temporal range query may have various forms depending on the mutability of objects and queries (i.e., stationary queries on moving objects, moving queries on stationary objects and moving queries on moving objects). In addition, a query may ask about the past, present, or the future.

1.2 Accuracy and Performance Tradeoff

In location-aware environments real-time or near real-time response is crucial, as delaying the query results may make them obsolete and thus useless by the time the system sends the response. However, producing fast results may come at a price of accuracy. More accuracy requires more processing power, more computation and more memory resources. With an extremely large number of location updates arriving from objects and queries via data streams and with typically limited system resources (e.g., memory) producing real-time answers presents a problem of an accuracy-performance tradeoff.

Applications have very diverse requirements on location-based services. For some a fast re-

sponse time is essential even at the cost of some loss of accuracy, while others require the highest accuracy even if the delivery of such accurate results comes at the loss of real-time response.

An example, where lower accuracy is acceptable is cellular companies continuously retrieving all cellular users entering and leaving the regions serviced by their towers. If the density of cellular users at any particular area significantly increases, they can automatically increase the bandwidth and thus provide a better service. In this case the accuracy is not of such a big concern. Even if the count of the number of users is approximate, it would not affect the overall utility of the results.

Other applications demand the highest accuracy possible even with limited system resources. In an emergency case scenario, such as, a gas spill, for example, one may want to know which people (and for how long) have been inside the allegedly contaminated area. Police, fire-fighters, and paramedics, the so-called first responders, would try to decontaminate victims and treat them with antidotes. They would establish a hot zone where contamination is highest. In the case of fast-acting nerve agents, antidotes need to be given within a half hour of the attack to be effective, so accuracy plays a key role. Knowing the locations and the times of when and for how long people were inside the affected region would be critical in providing early detection and appropriate treatment because of their exposure to toxic chemicals. This can be very critical as many would be at risk of developing leukemia, immune dysfunction, anemia or lung cancer - the diseases that can take years to diagnose, but once discovered at a later stage can be fatal.

The examples above illustrate that some applications emphasize getting the results as quickly as possible even with some loss in accuracy, while others prefer accuracy at the cost of the performance. This performance-accuracy tradeoff question calls for the evaluation of the common approaches in location modelling in spatio-temporal algorithms and determine which model would be more appropriate than the other for any spatio-temporal query system under certain conditions.

A number of works have been proposed for efficient evaluation of continuous spatio-temporal queries. Most of them focus on ways to reduce time and memory consumption, improve performance and increase scalability, by studying, for example, indexing techniques [35, 66, 65, 77], shared execution paradigm [62, 57, 56, 84], variations of algorithms [71, 70, 74, 60]. These exist-

ing solutions, however, tend to forego the choice of location modelling, seemingly randomly make a choice, often without justifying its affect on accuracy and performance when evaluating spatio-temporal queries. This thesis addresses this shortcoming by studying the performance-accuracy tradeoff question using the common approaches for location modeling in spatio-temporal algorithms. In this thesis, I investigate how the key factors, such as location update probability (or in other words, frequency of updates) and speed of objects and queries affect performance, accuracy or both when using one model or the other. I use these factors to characterize the tradeoff between performance and accuracy. This then serves as foundation to guide the selection when one model would be more appropriate than the other for the design of future spatio-temporal query system. This could even be utilized for dynamically adapting among different choices of location modelling behavior inside a spatio-temporal query engine. A system could dynamically switch between the two models to maximize the accuracy and minimize the cost of execution.

1.3 Algorithms for Processing Continuous Queries on Moving Objects

Many recent research works try to address this problem of efficient evaluation of continuous spatio-temporal queries. Some focus on indexing techniques [35, 66, 77], other on shared execution paradigms [62, 57, 84], or variations of algorithms [71, 60]. A major shortcoming of these existing solutions, however, is that most of them still process and materialize every location update individually. Even in [57, 84] where authors exploit *shared execution* paradigm, when performing a join, each moving object and moving query location update is processed individually. With an extremely large number of objects and queries, many comparisons must be made, consequently increasing the processing time tremendeously. In this paper I propose to exploit motion clustering as means to compress data and thus improve performance. In particular, I explore the idea of organizing data in such a way as to minimize memory requirements without any or only very little loss of information. Moreover, the execution is orchestrated in such a way as to reduce the number of

unnecessary spatial join executions between objects and queries.

In many applications moving objects are naturally in groups, in other words in clusters, including traffic jams, animal and bird migrations, groups of children on a trip or people evacuating from danger zones (e.g., fire, hurricanes). These moving objects often have some common properties, that makes clustering suitable. Clustering analysis, which groups similar data to reveal overall distribution patterns and interesting correlations in datasets, is useful in a number of applications, including data compression, image processing, and pattern recognition [58, 89].

In [89] Zhang et. al. exploited *micro-clustering* i.e., grouping data that are so close to each other that they can be treated as one unit. In [54] Li et. al. extended the concept to *moving micro-clusters*, groups of objects that are not only close to each other at a current time, but also likely to move together for a while. In this thesis, I utilize the concept of *moving micro-clusters*[1] as a way to abstract the data and optimize the execution of spatio-temporal queries. This serves as the means to improve the performance and achieve scalability when executing continuous spatio-temporal queries.

1.4 Clustering Algorithm for Moving Objects

I propose the *Scalable Cluster-Based Algorithm* (SCUBA) for evaluating continuous spatio-temporal queries on moving objects. SCUBA exploits a *shared cluster-based execution* paradigm, where moving objects and queries are grouped together into clusters based on common spatio-temporal attributes. Then execution of queries is abstracted as a *join-between* clusters and *join-within* clusters. In *join-between*, two clusters are tested for overlap (i.e., if they intersect with each other). In *join-within*, objects and queries inside clusters are joined with each other. By using clusters, we achieve data compression (i.e., organizing data in such a way as to reduce the total amount of data) and thus savings in memory consumption. This makes the evaluation of continuous queries more efficient. The individual positions of cluster members are represented in one of the following

[1] I use the term *moving clusters* in this paper

ways: (1) The positions of all cluster members are maintained relative to the centroid. This can be described as *lossless* data compression; (2) Individual positions are ignored, and the cluster itself is used to approximate locations of cluster members; (3) Relative positions are maintained for only a subset (e.g., the furthest from the centroid) of the cluster members. The rest of the members are abstracted into a nested (inside the moving cluster) structure called *nucleus*, and their relative positions are not preserved. Both moving clusters and nuclei serve as data compression structures, which approximate positions of their members in a compact form.

SCUBA introduces a general framework for processing large numbers of simultaneous spatio-temporal queries. Similar to [57, 84], SCUBA is applicable to all mutability combinations of objects and queries: (1) Stationary queries issued on moving objects. (2) Moving queries issued on stationary objects. (3) Moving queries issued on moving objects. When moving clusters are formed, they can be treated just as regular moving objects. Thus the existing algorithms and indexing techniques can be easily reused by extending them to moving clusters.

SCUBA employs a shared-cluster based execution where moving clusters (containing moving objects and queries) are periodically (every Δ time units) joined with each other. Since the processing is first done at an abstract level (at the level of moving clusters), if the clusters don't satisfy a join condition (i.e., don't overlap), the objects and queries belonging to these clusters don't need to be joined individually. Of course, maintaining clusters, which includes forming, dissolving, and expanding, comes with a cost. But our experimental evaluations demonstrate it is much cheaper than keeping the complete information about individual locations of objects and queries and processing them individually.

For simplicity, I present SCUBA in the context of continuous spatio-temporal range queries. However, SCUBA is applicable to other types of spatio-temporal queries, including *knn* queries, trajectory and aggregate queries.

1.5 Contributions

This thesis contributes to the advancement of spatio-temporal query processing in streaming databases in the following ways:

- I use the concept of moving clusters to abstract moving objects' and queries' based on common spatio-temporal attributes.
- I propose SCUBA - a scalable cluster based algorithm for evaluating a large set of continuous queries over spatio-temporal data streams.
- I utilize the shared cluster-based execution as means to achieve scalability for continuous spatio-temporal queries on moving objects.
- I present an analytical evaluation of SCUBA in terms of memory requirements, number of join comparisons and I/O cost.
- I implement SCUBA, discrete and continuous location modelling algorithms within the stream processing system CAPE [64].
- I provide experimental evidence that SCUBA improves the performance when evaluating spatio-temporal queries on real data generated by Network-Based Moving Objects Generator [10] in the city of Worcester, USA.
- I provide a tradeoff analysis of discrete and continuous location modelling algorithms in terms of their overall performance and accuracy. This then serves as foundation to guide the selection when one model would be more appropriate than the other for any spatio-temporal query system.
- I present an accuracy model for comparing discrete and continuous location modelling query results, which makes it possible to compute accuracy for otherwise very different types of query answers.

- I have conducted a comprehensive set of experiments assessing the performance vs. accuracy tradeoff of the two alternate location modelling strategies, again based on real data generated by Network-Based Moving Objects Generator in the city of Worcester, USA.

1.6 Organization of the Thesis

This thesis is divided into four parts, which are comprised of seven chapters as well as an appendix.

Part I:

- *Chapter 1* contains this introduction.
- *Chapter 2* contains an overview of past achievements and related work in spatio-temporal query processing, as well as clustering analysis.

Part II:

- *Chapter 3* provides general scalable architecture of the spatio-temporal operator inside CAPE exploiting shared, grid-based execution and implementing the two alternate location modelling techniques, discrete and continuous. It also contains a discussion of the pros and cons of the discrete and continuous models and describes their implementation inside CAPE in detail.
- *Chapter 4* describes our method for measuring accuracy between the discrete and continuous models. In addition, several scenarios are presented here to illustrate the model in use.
- *Chapter 5* contains our experimental evaluations and tradeoff analysis of discrete and continuous algorithms in terms of their overall performance and accuracy.

Part III:

- *Chapter 6* discusses the clustering algorithms, and my preferred choice using set of criteria supported by literature.

- *Chapter 7* introduces a scalable cluster-based algorithm SCUBA and contains theoretical analysis of SCUBA in terms of memory requirements, join cost and I/O cost.

- *Chapter 8* provides experimental evaluations of SCUBA and the conclusions about efficiency of the algorithm.

Part IV:

- *Chapter 9* concludes and describes possible extensions of this research work.

- *Appendix A* contains a UML diagram portraying the classes that the implementation consists of. It also contains lists of the functions that these classes contain.

Chapter 2

Related Work

In this chapter I will briefly discuss some areas of related work in streaming databases, spatio-temporal databases and moving objects databases. This related work serves as a starting point for creating our own framework for evaluating continuous spatio-temporal queries on moving objects.

2.1 Data Stream Management Systems

First, I am going to discuss the related work in streaming databases, since the continuous spatio-temporal query processing is a part of the streaming data processing paradigm. The key difference between a classical Database Management System (DBMS) and a Data Stream Management System (DSMS) is the *data stream model*. Instead of processing a query over a persistent set of data that is stored in advance on disk, queries are performed in DSMSs over a data stream. In a data stream, data elements arrive on-line and stay only for a limited time period in memory. Consequently, the DSMS has to handle the data elements before it runs out of memory. The order in which the data elements arrive cannot be controlled by the system. Once a data element has been processed it cannot be retrieved again without storing it explicitly. The size of data streams is potentially unbounded. In DSMSs, *continuous queries* are evaluated over continuously arriving data elements. Since continuous streams may not end, intermediate results of continuous queries are often generated over a predefined window and then either stored, updated, or used to generate

a new data stream of intermediate results [3]. Window techniques are especially important for aggregation and join queries. Examples for DSMSs include STREAM [3], GigaScope [19], Aurora [1], NiagaraCQ [16], CAPE [64], Nile [37] and TelegraphCQ [12].

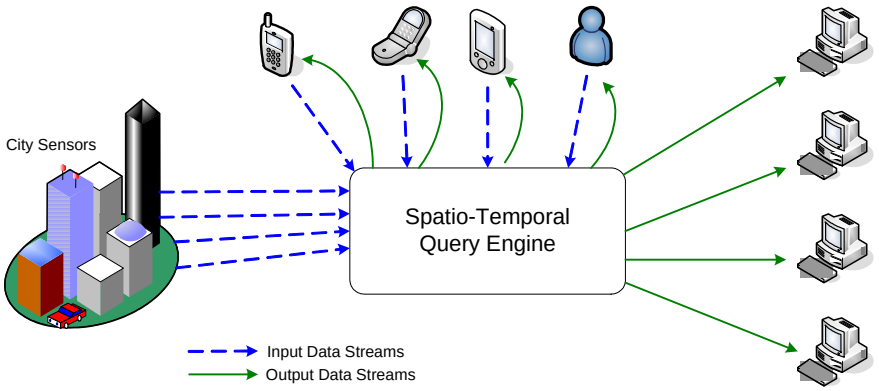


Figure 2.1: Spatio-temporal query engine

2.2 Continuous Spatio-Temporal Queries

The growing importance of moving object environments is reflected in the recent body of work addressing issues such as indexing, uncertainty management, and models for spatio-temporal data. Different indexing techniques have been proposed for moving objects in the literature e.g., [7, 49] index the histories, or trajectories, of the positions of moving objects, while [66] indexes the current and anticipated future positions of moving objects. In [48], trajectories are mapped to points in a higher-dimensional space that are then indexed. In [66], objects are indexed with the index structure parameterized with velocity vectors so that the index can be used at future times. This is achieved by assuming that an object will remain at the same speed and in the same direction until an update is received from the object. A similar assumption about the moving objects' updates is made in this thesis.

Uncertainty in the positions of the objects is dealt with by controlling the update frequency [61, 81], where objects report their positions and velocity vectors when their actual positions deviate

from what they have previously reported by some threshold. Tayeb et. al. [77] use quadtrees for indexing moving objects. Kollios et. al. [48] map moving objects and their velocities into points and store the points in KD-tree. Pfoser et. al. [61] index the past trajectories of moving objects that are presented as connected line segments. The problem of answering a range query for a collection of moving objects is addressed in [2] through the use of indexing schemes using external range trees. [80, 82] consider the management of collections of moving points in the plane by describing the current and expected positions of each point in the future. They address how often to update the locations of the points to balance the costs of updates against imprecision in the point positions. Spatio-temporal database models to support moving objects, spatio-temporal types and supporting operations have been developed in [24, 32].

2.2.1 Indexing Moving Objects/Queries

Many of the existing spatio-temporal index structures [22] aim to modify the traditional R-tree [33] to support the highly dynamic environments of location-aware servers. In particular, two main approaches are investigated: (1) Indexing the future trajectories such that the existing tree would last longer before an update is needed. Examples of this category are the TPR-tree [66], REXP -tree [65], and the TPR*- tree [73]). (2) Modifying the deletion and insertion algorithms for the original R-tree to support frequent updates. Examples of this category include the Lazy update R-tree [50] and the Frequently-updated Rtree [53]. However, even with the proposed modifications of the R-tree structures, highly dynamic environments degrade the performance of the R-tree and result in a bad performance. In our system, we thus avoid using R-tree-like structures. Instead, I opt to make use of a grid-like index structure [57] that is simple to update and retrieve. Moreover, fixed grids are space-dependent, thus there is no need to continuously change the index structure with the continuous insertion and deletion. Several spatio-temporal systems [62, 57, 65] utilize grid index for indexing moving objects.

2.2.2 Scalability

Most of spatio-temporal queries are continuous in nature and require continuous evaluation as the query result becomes invalid with the change of information [78]. One way to handle continuous queries is to abstract them into a series of snapshot queries executed at regular time intervals, i.e., periodically. Existing algorithms for continuous spatio-temporal queries aim to optimize the time interval between the periodic executions. Three different approaches are investigated: (1) The validity of the results [88, 90]. With each query answer, the server returns a valid time [90] or a valid region [88] of the answer. Once the valid time is expired or the client goes out of the valid region, the client re-submits the continuous query for re-evaluation. (2) Caching the results. The main idea is to cache the previous result either on the client side [68] or in the server side [52]. Previously cached results are used to optimize the search for the new results of k-nearest-neighbor queries [68] and range queries [52]. (3) Pre-computing the result [52, 72]. If the trajectory of query movement is known in advance, then by using computational geometry for stationary objects [72] or velocity information for moving objects [52], we can identify which objects will be nearest-neighbors [72] to or within a range [52] from the trajectory of the query.

2.2.3 Variety of Queries

Most of the existing query processing techniques focus on solving special cases of one type or category of spatio-temporal queries. For example, [68, 72, 88, 90] are valid only for moving queries on stationary objects. Whereas, [11, 26, 34, 62] are valid only for stationary range queries on moving objects. Other works focus on aggregate queries [34, 69, 70] or k-NN queries [44, 68]. Trying to support a wide variety of continuous spatio-temporal queries in a location-aware server presents a challenge as it forces implementation of a variety of specific algorithms with different access structures. We try to avoid this by designing a generic and flexible structure of a spatio-temporal operator that allows for easy extension and integration of different query types. Even though this thesis is presented in the context of spatio-temporal range queries, the existing work was designed to be as generic as possible for easy integration of various query types.

2.2.4 Large Number of Queries

Many of the existing spatio-temporal algorithms focus on evaluating only one spatio-temporal query. In a typical location-aware server, there is a huge number of concurrently executing continuous spatio-temporal queries. Handling each query as an individual entity dramatically degrades the performance of the location-aware server and in some situation can become prohibitive. There is a lot of research in sharing the execution of continuous web queries (e.g., see [16]) and continuous streaming queries (e.g., see [13, 14, 36]).

Optimization techniques for evaluating a set of continuous spatio-temporal queries are recently addressed for centralized [62] and distributed environments [11, 26]. The main idea of [11, 26] is to ship part of the query processing down to the moving objects, while the server mainly acts as a mediator among moving objects. In centralized environments, the Q-index [62] is presented as an R-tree-like [33] index structure to index the queries instead of objects. However, the Q-index is limited as it is applicable only for stationary queries. Moving queries would spoil the Q-index and hence dramatically degrade its performance.

Another popular method for supporting large number of queries that has been exploited is the idea of *shared execution*. The shared execution has been used in NiagaraCQ [16] for web queries, in PSoup [14] for streaming queries, in SINA [57] for continuous spatio-temporal range queries, and in SEA-CNN [84] for continuous spatio-temporal kNN queries.

One major shortcoming of these existing solutions, however, is that many of them still process and materialize every location update individually. When performing a join, each moving object and moving query location update is processed individually. With an extremely large number of objects and queries, many comparisons must be made, consequently increasing the processing time tremendously.

2.2.5 Clustering Analysis

Clustering is a well-studied area in mathematics and computer science. It is related to many different areas including classification, databases, data-mining, spatial range-searching, etc. As such, it

has received a lot of attention. Some of the works include [21, 31, 47, 63, 20, 86]. For an elaborate survey on clustering, readers are referred to [46]. Most of the previous work focuses on using clustering to analyze static or dynamic data and find interesting things about it. Instead, I now propose to utilize clustering idea as a means to abstract data (to enable cluster-based load shedding) and achieve scalable processing (minimize processing time) of continuous queries on moving objects. To the best of our knowledge, this is the first work to use clustering for internal optimization of continuous query processing on streaming spatio-temporal data.

Clustering Data Streams

The commonly used clustering algorithm for offline (or non-incremental) clustering, *K-means*, is described in [21, 63] and introduced in more detail in Chapter 7. The objective of the algorithm is to minimize the average distance from data points to their closest cluster centers. An alternative interpretation of K-median clustering is that we would like to cover the points by k balls, where the radius of the largest ball is minimized [38].

Given a sequence of points, the objective of [28] Guha et. al. is to maintain a consistently good clustering of the sequence observed so far, using a small amount of memory and time. They give constant-factor approximation algorithms for the K-median problem in the data stream model in a single pass. They study the performance of a *divide-and-conquer* algorithm, called *Small-Space*, that divides data into pieces, and then again clusters the centers obtained (where each center is weighted by the number of points closer to it than to any other center). The authors also propose another algorithm (*Smaller-Space*) that is similar to the piece-meal approach except that instead of re-clustering only once, it repeatedly re-clusters weighted centers [28]. The advantage of Small(er)-Space is that we sacrifice somewhat the quality of the clustering approximation to obtain an algorithm that uses less memory. Their model and and analysis have similarities to incremental clustering and online models. However, the approach is a little bit different. They maintain a "forest" of assignments. They complete this to k trees, and all the nodes in a tree are assigned to the median denoted by the root of the tree. The disadvantage of this algorithm is similar to that of

K-means, namely the number of clusters must be known in advance (the number of clusters is the input parameter to the K-means algorithm).

Competitive Learning Clustering and the basic *Leader-Follower Clustering* are two algorithms for online (i.e., incremental) clustering presented in [21]. John A. Hartigan had already proposed the latter in an early publication on clustering algorithms [39]. One of the disadvantages of the Leader-Follower Clustering algorithm is that it lacks the ability to keep the number of clusters constant, so a large number of clusters might be created (potentially as many as there are data points). But this disadvantage of the Leader-Follower algorithm could actually be easily addressed by merging the clusters, if appropriate. Competitive Learning Clustering can be transformed in a single-scan algorithm to save the clustering time. In its basic form, however, it depends on a convergence criterion that makes several iterations over the data necessary. Other sources also describe these two algorithms, but name them differently. In [51] they are named as *Growing K-means Clustering* and *Sequential Leader Clustering*[1].

In [43], the author uses a clustering algorithm that pre-processes data points that arrive each second. Based on the clustering model, a Markov Model is learned and finally used for a prediction task. They used the k-means algorithm to pre-process data. The final result is an algorithm that is capable of clustering streaming data and learn a Markov Model with one scan over the data set. It is called the *Extended Leader-Follower* algorithm (*ELF* algorithm). Again, because the K-means algorithm requires the number of clusters to be known in advance, and with each data point update the clustering might change, this approach doesn't work well for very dynamic data points that represent the location updates of moving objects and queries.

A list of considerations and criterions when dealing with incremental data stream algorithms is given in [20, 86, 6]. Incremental algorithms should only use a small and constant amount of memory. Consequently, a compact representation of the current model is accessible at all time. The running time and hence the computational complexity should be such that new incoming data points can be processed at their arrival. The algorithm should be capable of distinguishing between

[1] In this thesis, I refer to these algorithms as *Competitive Learning* clustering and *Leader-Follower* clustering.

outliers, emerging patterns and noise.

Barbara in [6] presents an overview of the clustering algorithms *BIRCH* [89], *COBWEB* [23], *STREAM* [28, 59], and *Fractal Clustering* [5], which all could be used for an incremental clustering of data streams. He describes the advantages and shortcomings of these algorithms with respect to compactness, functionality and outliers.

BIRCH clusters data points using a CF-tree - a height-balanced tree (analogous to a B-Tree). One of the drawbacks of the BIRCH method is that after some time it draws into secondary memory. And even though it tries to minimize the number of I/Os for clustering a new point, it still takes a considerable amount of time to do so [6]. The processing is better done in batches to try to amortize the overall cost.

COBWEB [23] implements hierarchical clustering via a classification tree. The classification tree is not height-balanced which often causes the space (and time) complexity to degrade dramatically. This makes COBWEB an unattractive choice for data streams clustering.

STREAM [28, 59] aims to provide guaranteed performance by minimizing the sum of the square of the distances of points to the centroids (similar to K-means). STREAM processes data streams in batches of points by first clustering the points in each batch and then keeping the weighted cluster centers (i.e., the centroids weighted by the number of points attracted to them). Then STREAM clusters the weighted centers to get the overall clustering model. If the batches are of equal size, the first clustering iteration has a constant processing time. But for the second iteration of clustering the time can increase without bounds as more batches of data arrive. Moreover, it is recognized by its authors [59] that it takes longer than K-means to find a bounded solution.

Fractal Clustering (FC) [5] groups points that show self-similarity, by placing them in the cluster in which they have the minimal fractal impact. FC works with several layers of grids (the cardinality of each dimension is increased 4 times with each next layer), and even though only occupied cells are kept in memory, the method suffers from high memory usage [8].

Gaber et. al. in [25] have proposed algorithms for incremental clustering, a simple so-called *one-look* clustering algorithm that takes into account the available resources of a machine. In [15]

Chaudhuri presents various considerations for clustering algorithms, such as which actions are possible or necessary when new data points are added to an existing model.

Clustering Motion

The difficulty in maintaining and computing clusters on moving objects is the underlying kinetic nature of the environment [29]. Once the clusters are computed at a certain time, and the time progresses, the clustering may change and deteriorate. To remain a "high quality" clustering (i.e., the cluster sizes are small compared to the size of the optimal clustering) one needs to maintain the clustering by either reclustering the points every once in a while, or alternatively, move points from one cluster to another. The number of such "maintenance" events may dominate the overall running time of the algorithm, and the number of such events can be extremely large, thus hampering the processing time.

In [89] Zhang et. al. described *micro-clustering* i.e., grouping data that are so close to each other that they can be treated as one unit. In [54] Li et. al. extended the concept to *moving micro-clusters*, groups of objects that are not only close to each other at a current time, but also likely to move together for a while.

In [38], authors analytically study motion clustering. They define the clustering motion problem as following: Let $P[t]$ be a set of moving points in $\Re^d$, with a degree of motion μ; namely for a point $p[t] \in P[t]$, we have $p(t)$=($p_1(t)$,...$p_d(t)$), where $p_j(t)$ is a polynomial of degree μ, and t is the time parameter, for $j = 1, ..., d$. The authors in [38] demonstrate that if one is willing to compromise on the number of clusters used, then clustering becomes considerably easier (computationally) and it can be done quickly. Furthermore, we can trade off between the quality of the clustering and the number of clusters used. Hence, one can compute quickly a clustering with a large number of clusters, and cluster those clusters in a second stage, so to get a more reasonable k-clustering. The authors also propose an algorithm for picking a "small" subset of the moving points by computing a fine clustering, and picking a representative from each cluster. The size of the subset, known as *coreset*, is independent of *n* (number of data points), and it represents the

k-clustering of the moving points at any time. Namely, instead of clustering the points, only the representative points get clustered. This implies that one can construct a data structure that can report the approximate clustering at any time.

Chapter 3

Part II:

Accuracy vs. Performance Tradeoff in Location-Based Services

3.1 Spatio-Temporal Operator

Query plans in database systems are composed of operators, which perform the actual processing, such as my spatio-temporal operator. In this chapter, I discuss the design of my spatio-temporal operator inside the CAPE data streaming system. The operator utilizes a grid index and implements discrete and continuous location modelling techniques. I begin by first introducing the continuous and discrete location models, and then proceed with the operator design that implements these two models.

3.2 Modelling Motion

People move through space. Different types of movement occur, such as the movement of a person who walks, runs, or rides a bus, a taxi that travels through city, or an oil spill spreading on the water.

Although these movements occur continuously in reality, people conceptualize certain movements as being discrete, while others are conceptualized as being continuous [85, 42]. Each model has its own advantages and disadvantages as I am going to discuss next.

3.2.1 Discrete Model

Many existing location-based services [57, 84] model the location as a discrete position in time. For each moving object, a position point of the form (x, y, z, t) is generated periodically, indicating that the object is at location x, y, z at time t. This point-location management [79] has several advantages. The computations involving points are relatively simple and thus fast. This allows for a speedy processing of a extremely large number of location updates. This facilitates scalability in terms of the number of concurrent continuous spatio-temporal queries and the number of moving objects sending their updates to the query system. Since location updates are processed as they come and retained in the system for a relatively short period of time, on average the point-based systems require less time than for example the trajectory-based systems. So for applications that require very fast results, a discrete model suits pretty well.

However, at the same time the discrete model has several critical weaknesses. First, the method does not enable interpolation or extrapolation. For example, assume that a gas company dispatcher needs to know "*Which service crew was within ten miles from the location of a house gas leakage that occurred at 8PM*". This information can only be retrieved for moving objects that happened to send a location update at 8PM. If the system didn't receive a position with an 8PM timestamp, then the whereabouts of the object at that time are unknown. Thus the discrete system cannot answer such a query. The problem is even more severe if a future location is requested (e.g., *Which ambulance will be the closest to the scene of the disaster in the next 30 minutes*?) This query cannot be answered by the discrete location method.

Another problem with the discrete point-location model is that it has a higher likelihood of giving incorrect results. Consider a server periodically executing queries on a set of discrete location updates [57]. If a moving object location update (O_1, pos_1, t_1) is received first and immediately

after it a moving query location update (Q_1, pos_1, t_2), it might calculate that an object is inside the query and return a positive result. Potentially this might be an incorrect result, if at time t_2 the object, in fact, has moved outside of the boundaries of the query Q_1, but we have not yet received its new location update.

The third problem that might arise when utilizing the discrete method is that it can lead to a precision/resource trade-off. An accurate picture of the precise location of moving objects would require frequent location updates. This consumes precious resources such as bandwidth, processing power and memory resources [67], since more frequent updates would need to be processed to get an accurate picture of the movement. Moreover, the server would have to send the results more frequently. With an extremely large number of objects and queries, this may create a performance bottleneck.

Lastly, a point-based discrete model is incapable in answering queries related to time intervals, such as "*Give me all airplanes that entered the turbulent region in the Pacific and have been inside it for more than 30 minutes*".

In general, discrete systems focus on distinct positions of objects at a time instance, and thus are limited in answering queries related to time durations. Typically, they process data without extracting any additional information (e.g., speed, direction, acceleration, distance travelled, etc.). This additional information often might be helpful in processing dynamic location data more efficiently and in maximizing accuracy which is what continuous models of location tracking attempt to exploit.

3.2.2 Continuous Model

A trajectory model is a more sophisticated model, which solves several of the problems that arise in discrete location modelling. Beyond that it offers some additional benefits that I am going to mention in this section. For simplicity, I assume linear trajectories. However, other trajectory models can be used, such as curve-based [87], spline interpolation [55], and motion functions [27].

We define a trajectory as a sequence of straight-line segments (x_1, y_1, t_1), ... ,(x_n, y_n, t_n) in a

3-dimensional space. A linear trajectory means that when the object starts at a location having coordinates (x_1, y_1) at time t_1, it will move on a straight line at constant speed and will reach location (x_2, y_2) at time t_2, and so on. This type of trajectory is an approximation of the expected motion of the object in space and time. The object may not move in straight lines at a constant speed. However, given enough location points, the approximation can be accurate up to an arbitrary precision. This brings us to the accuracy-performance tradeoff question. The number of line segments in the trajectory has an important implication on the performance of the system. Specifically, the performance increases and the accuracy decreases as the number of line segments decreases.

Using trajectories we can compute the expected location of the moving object at a time t. This technique enables both location interpolation and extrapolation. We can compute the expected location of the moving object at any point in time without receiving an explicit update. In addition, unlike the discrete approach, using trajectories we can answer queries related to time intervals, and compare the trajectories of moving objects and queries.

Another advantage is that one can associate an arbitrary uncertainty threshold with the trajectory and if arriving location updates are within that threshold from the trajectory, it can be assumed that the trajectory is a good approximation. This agreement (the trajectory plus the uncertainty threshold) between the moving object and the server solves the problem of the tradeoff between resource/bandwidth consumption and accuracy [83]. In the trajectory model, if the moving object does not deviate from its prescribed trajectory by more than the uncertainty threshold, we can discard the incoming position updates. Thus we save on memory consumption and, to some degree, in computation of the join. An additional bonus of the continuous model is that the answers are also represented as 3-D trajectories (i.e., indicating how long an object satisfied a particular query). This may limit the amount of data that needs to be sent and saves network bandwidth.

3.3 Preliminary

As a preliminary, I first describe the assumptions and restrictions in the system and the format of location updates that we expect arriving via data streams.

3.3.1 Spatio-Temporal Data Streams

We assume that moving objects and moving queries send their location updates periodically to the CAPE system. A location update from the client (moving object) to the server has the format(*OID*, *Pos*, $\{attr_1, attr_2 . . . attr_n\}$), where *OID* is the object identifier, *Pos* is the latest location of the object, and $\{attr_1 . . . attr_n\}$ are additional attributes describing the object (e.g., police car, fire-fighter, deer, etc.).

Once an object stops moving (e.g., an object reaches its destination or the device gets turned off) it sends a disappear message to the server which indicates that the object is no longer moving, and will no longer send any location updates.

3.3.2 Assumptions and Restrictions

Below is the list of current assumptions and limitations:

- Both objects' and queries' location updates arrive via data streams.
- All location updates (trajectories) fit in memory.
- Objects and queries move in straight lines and at a constant speed.
- All location updates always arrive in a strict sequential order (i.e., time stamps of the updates are increasing).
- Queries are answered based on the up-to-date knowledge.
- We assume both objects and queries move continuously and send a disappear messages when they get turned off.

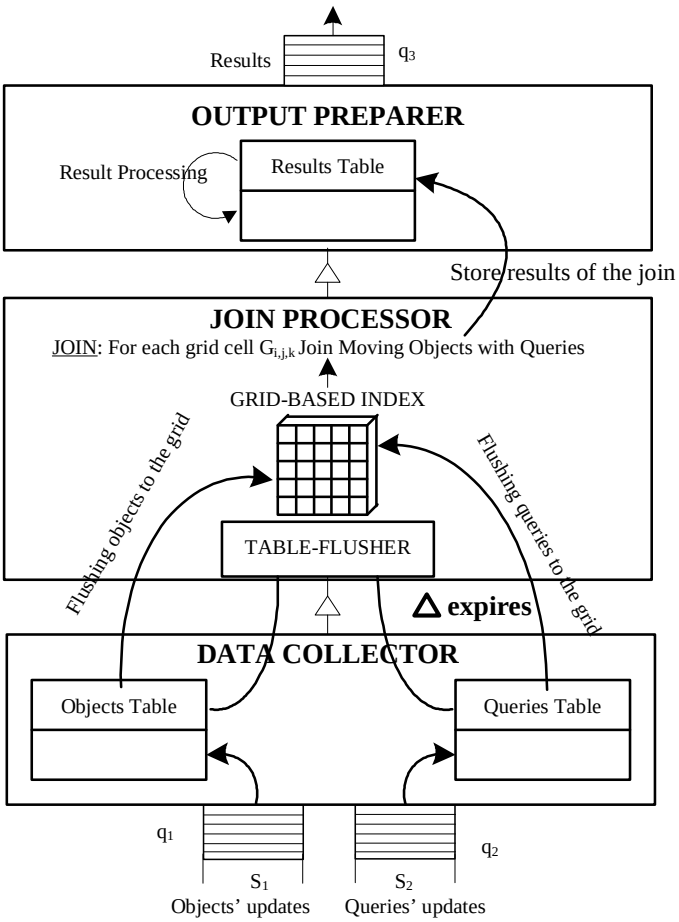


Figure 3.1: Architecture of spatio-temporal operator

- For the continuous model, the operator maintains the last two location updates for each object and query after Δ evaluation interval expiration. This is used to predict the movement of the object/query for the next time interval, even if no explicit update is received.
- Grid cell sizes and count of the grid index are fixed and pre-determined in advance.

3.4 Shared Execution Architecture Overview

Next I describe the main components of the motion operator utilizing shared execution similar to [26, 35, 62, 57, 84], where spatio-temporal queries are grouped together and the execution is abstracted as a spatial join between moving objects and queries.

3.4.1 General Design

The spatio-temporal operator has three components: *Data Collector*, *Join Processor*, and *Output Preparer* (Figure 3.1).

Data Collector (DC). The data sources are streams S_1 and S_2 transmitting location updates for objects and queries respectively. The incoming tuples are buffered up in the corresponding queues q_1 and q_2. The Data Collector (Algorithm 1) periodically reads from the queues. It processes the location updates and based on the entity type materializes them into Objects and Queries tables correspondingly.

Algorithm 1 *DataCollector()*

```
1:  loop
2:     Dequeue tuples from queue q1 (Moving objects stream)
3:     while there are unprocessed objects' tuples do
4:        for every object location update tuple o do
5:           Find object entry in Objects Table using OID.
6:           Increment count for number of location updates for o.
7:           Update location information for object o.
8:        end for
9:     end while
10:    Dequeue tuples from queue q2 (Moving queries stream)
       ... //Similar processing as moving objects
11: end loop
```

Join Processor (JP). Once Δ time interval expires, the Join Processor (Algorithm 2) activates its submodule - *TableFlusher* (TF). TF iterates over the *Objects Table* and grabs all objects' updates and flushes them into the shared grid index structure. Similarly the queries are flushed based on their location updates from *Queries Table* into the grid. Then the Join Processor initiates a join algorithm which iterates over each grid cell, performing a join with all objects against all queries that had been indexed into that particular grid cell. If they intersect, the join algorithm reports the output to the *Results Table*.

Output Preparer (OP). The Output Preparer (Algorithm 3) iterates over the Results table and hashes each result pair by query id and object id (*QID*, *OID*), to organize all answers related to one pair into an output result structure. If desired, the output preparer can further sort the results by the output timestamps to guarantee order. Finally, the output preparer places the output tuples into the

Algorithm 2 *JoinProcessor()*

```
1: Activate TableFlusher
2: TableFlusher: Insert moving objects and moving queries locations into grid
3: for i = 0 to MAX_X_DIM do
4:    for j = 0 to MAX_Y_DIM do
5:       for t = 0 to MAX_T_DIM do
6:          for every query q in grid cell G_{i,j,t} do
7:             for every object o in grid cell G_{i,j,t} do
8:                R = DoJoin(q,o,G_{i,j,t}) //join moving queries with moving objects
9:                Insert results R into Results Table
10:            end for
11:         end for
12:      end for
13:   end for
14: end for
15: Call OuputPreparer() //Initiate OutputPreparer component
```

output queue q_3 of the operator.

Algorithm 3 *OutputPreparer()*

```
1: for every result entry R in Results Table do
2:    Hash each result pair by (QID,OID)
3: end for
4: for every result pair (QID,OID) do
5:    Sort results
6:    if continuous location modelling then
7:       Interpolate results (construct continuous segments out of discrete points)
8:       Merge results (merge the segments by the common time stamps)
9:    end if
10:   Put results into output queue q_3
11: end for
```

3.5 Data Structures

During the course of the execution, the operator maintains the following data structures:

- **Objects table**. Objects are organized within the in-memory table. The object entry has the form of (*OID, Pos,* $\{attr_1, attr_2 \ldots attr_n\}$), where *OID* is the object identifier, *Pos* is the latest location of the object, and $\{attr_1 \ldots attr_n\}$ are additional attributes describing the object.

- **Queries table**. Queries similar to objects are organized within the in-memory table. The

query entry has the form of (*QID*, *Pos*, $Bound_H$, $Bound_W$,{$attr_1$, $attr_2 \ldots attr_n$}). *QID* is the query identifier. If we assume rectangular regions, then *Pos* is the latest location of the query focal point. $Bound_H$ is the vertical distance and $Bound_W$ the horizontal distance from the focal point of the query to the edge of the query. $\{attr_1 \ldots attr_n\}$ are additional attributes describing the query.

- **Results table**. Results are temporarily stored in the in-memory Results table in the form of (*QID*, *OID*, *Intersect*). *Intersect* is an attribute describing the intersection between the object and the query.

- **In-memory grid**. The memory-based NxMxT grid is divided into NxMxT grid cells (where N and M are spatial dimensions, and T is a temporal dimension of the grid). Objects and queries are hashed based on their locations and the time of updates to the grid cells. For each grid cell the in-memory grid maintains a list of *OID*s and *QID*s of objects and queries respectively whose location updates were hashed into that cell. For queries we create an entry in each of the cells its region (for discrete) or volume (for continuous) overlaps with.

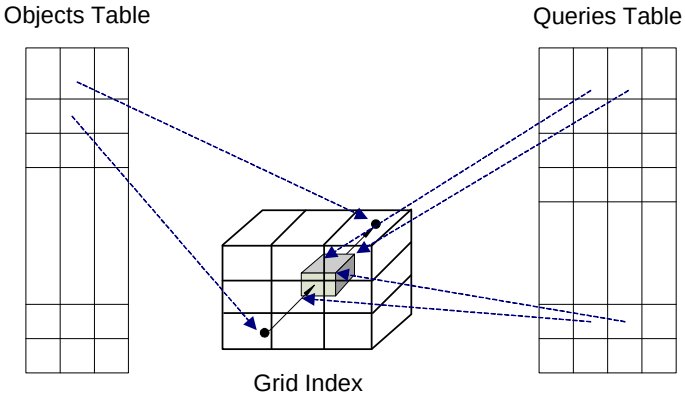


Figure 3.2: Data structures used by regular motion operator

Choosing an appropriate index for continuously moving objects and moving queries was critical for evaluating queries on moving objects with near real-time response requirement. Traditional

spatial index structures such as R-trees [33] are not appropriate for indexing moving objects because the location changes of objects may cause splitting or merging of the nodes constantly or even rebuilding the entire tree from time to time. The index structure must provide optimal updating performance.

I chose an in-memory grid index, since in the highly dynamic environment, maintaining a spatial index such as R-tree [33] on moving objects is not practical (as mentioned in the Related Work chapter). We expect the objects and the queries to frequently update their positions. This suggests that only memory based structures are suitable for the moving objects and queries table. We use grid cells to group moving objects because the grid structure is relatively inexpensive to maintain due to its static flat organization.

Observe also that the Objects and Queries tables keep location updates only for the duration of the time interval Δ. Once the Δ expires, all location updates received are flushed from the Objects and Queries tables respectively to the in-memory grid. After the results to queries are computed, the operator can either store the last known trajectories in the Objects and Queries tables (to be utilized in the next time interval) or clear out all the data. I exercise both choices; the former in continuous model of objects and queries, and the latter in discrete point-based model.

3.6 Discrete and Continuous Modes of Execution

I now introduce algorithms for processing continuous spatio-temporal queries by modelling objects and queries in discrete or continuous fashions, called *Discrete Scalable Point-Based Algorithm* (*DSA*) and *Continuous Scalable Trajectory-Based Algorithm* (*CSA*) respectively. The spatio-temporal operator can use either of the two algorithms for modelling moving entities.

Both DSA and CSA exploit the shared execution paradigm, similar to [57, 84] and abstract queries' execution by utilizing a grid-based spatial join between moving objects and queries. DSA describes moving objects and their locations as discrete points, and the queries as rectangular regions. CSA is more precise by describing the movement of objects via trajectories, and the

movement of queries via trajectory volumes. In the case of DSA, the predicate for join is the containment of a location point inside a rectangular region. In case of CSA it is an intersection between a trajectory and a trajectory volume. We re-evaluate the queries every Δ time units by computing the join between objects and queries. For DSA we assume that we don't know where objects and queries are unless we get an explicit location update. For both DSA and CSA we assume that location updates arrive in an ordered (by time) fashion. Therefore, we only join the location updates for objects and queries that arrived during the time interval Δ. Also for DSA, in order for an object to join the query, the timestamps for both have to match (i.e., if the object was inside a rectangular moving region at a certain location, the timestamps of both object and query have to be identical). For CSA, trajectories and trajectory volumes are constructed from the location updates that arrived before Δ expired. Trajectories (and trajectory volumes) allow us to abstract each moving entity location as a function of time $f(t)$. Using trajectories we can compute the location of the moving entity at any time during the time interval. For CSA we assume continuous movement. Even if no explicit location update has arrived, we assume the object is moving according to its latest trajectory.

In CSA, the answer is not just whether a particular object is a part of the result set for a particular query, but also the time of when it has occurred (and for how long). The results in CSA are polylines (functions of time) where the beginning of the polyline is the entering event (object entered query region) and the end is the leaving point (object left query region). This allows us to improve the accuracy of the query results, since we know not just that an object was inside the query, but also the spatial location and the time of when it entered it, and left it. One of the advantages of such an approach is that we can answer queries similar to "*Retrieve all objects that entered region R and stayed there for at least 3 minutes*".

3.6.1 Execution of DSA and CSA

The execution of DSA and CSA can be broken down into phases. DSA has three phases: (1) discrete position update, (2) joining, and (3) timeline phases. CSA has five phases: (1) the initial-

ization, (2) trajectory update, (3) joining, (4) merging and (5) timeline phases. The initialization phase is the first phase in CSA and has no corresponding equivalent in DSA. It initializes the expected positions of all the objects and queries in the system at the beginning of each time interval based on their last known trajectory. This is based on our assumption that all objects and queries move continuously in time. The trajectory update phase in CSA updates trajectories and trajectory volumes for moving objects and queries. The equivalent to it in the DSA is the discrete position update phase.

Both DSA and CSA have a joining phase, triggered every Δ time units. First, the objects and the queries are flushed into the in-memory grid index. Then a join is performed between the points and the rectangular regions (in case of DSA) and trajectory lines and the trajectory volumes (in case of CSA). The result of the joining phase in DSA is a set of points with the following format (*QID, OID, loc,* t_{loc}*,* t_{answer}), where *QID* and *OID* are the ids of the query and object, respectively. *loc,* t_{loc}*,* t_{answer} describe the intersection between the object and the query. *loc* is a discrete location update of the object, t_{loc} is the time of the location update, and t_{answer} is the time of query evaluation (since it might be delayed after the location update arrival at time t_{loc}). The joining phase in CSA results in a set of mini-clipped results. The mini-clipped results are segments of the original trajectory lines, indicating the parts of the trajectories intersecting with a query trajectory volume. The sizes of the mini-clipped results are constrained by the size of a query volume and the size of a grid cell.

After joining phase in CSA, the merging phase is triggered. The mini-clipped results from the joining phase are merged by *OID* and *QID* and starting and end positions of the mini-clipped results. There is no equivalent phase in the DSA for this. Finally, after merging is complete, the timeline phase orders the merged results by the starting timestamp to imitate the order of events in time. The same idea is behind the timeline phase in the DSA, where the results are ordered by t_{loc} timestamp. Ordered results are sent to the output stream, and if there are no other operators to process these tuples, then they are sent to the users expecting the results of the continuous queries.

Looking at the number and the functionality of the execution stages in algorithms, DSA re-

quires less processing time than CSA.

3.7 Analytical Observations

Below I analyze requirements needed for evaluating continuous queries utilizing either discrete or continuous models. I use parameters listed in Table 3.1 in the analysis.

Variable	Description
Δ	Time interval between periodic query execution
N_{obj}	Total number of objects in the system
N_{qry}	Total number of queries in the system
E_{obj}	Size of object entry in $Objects$ table
E_{qry}	Size of query entry in $Queries$ table
E_{grid}	Size of object/query entry in $GridIndex$
C_{AR}	Average number of grid cells that a query overlaps with
r_{obj}	Rate of arrival for moving objects' updates
r_{qry}	Rate of arrival for moving queries' updates
$T_{objects}$	Average number of mini-segments for object trajectory during time interval Δ
$T_{queries}$	Average number of mini-segments for query trajectory volume during time interval Δ
x_i	Position in the x-dimension
y_i	Position in the y-dimension

Table 3.1: Parameters used in analysis of regular grid-based shared execution of discrete and continuous location modelling techniques

I maintain three memory structures for both DSA and CSA, namely, the Objects table, the Queries table, and the NxMxT in-memory grid. In addition, the incoming tuples from the data streams are buffered up into Objects and Queries buffers before Δ time interval expires. During any time interval Δ, the memory size consumed by these structures is:

$$\begin{aligned} M = ObjectsQueue + QueriesQueue+ \\ ObjectsTable + QueriesTable + GridIndex \end{aligned} \tag{3.1}$$

For discrete model (DSA), memory requirements can be described as following:

$$
\begin{aligned}
M_{DSA} &= \Delta * r_{obj} * E_{obj} + \Delta * r_{qry} * E_{qry} + \\
&+ N_{obj} * E_{obj} + N_{qry} * E_{qry} + \\
&+ N_{qry} * C_{AR} * E_{grid}
\end{aligned}
\tag{3.2}
$$

where Δ *r_{obj}*E_{obj} and Δ *r_{qry}*E_{qry} are the objects' and queries' buffers respectively. N_{obj}*E_{obj} and N_{qry}*E_{qry} denote memory consumption by Objects and Queries tables, and N_{qry} *C_{AR} *E_{grid} is the memory consumed by the grid index. C_{AR} represents an average number of grid cells that a rectangular range query overlaps with. For the continuous model, the memory requirements could be characterized as following:

$$
\begin{aligned}
M_{CSA} &= \Delta * r_{obj} * E_{obj} + \Delta * r_{qry} * E_{qry} + \\
&+ N_{obj} * E_{obj} + N_{qry} * E_{qry} + \\
&+ 2 * E_{grid} * N_{obj} * T_{objects} + 2 * E_{grid} * N_{qry} * T_{queries};
\end{aligned}
\tag{3.3}
$$

The objects' and queries' buffers memory requirements of the continuous model are similar to the ones of the discrete. Similarly, Objects and Queries tables consume the same amount of memory as the discrete model. The difference is in the memory requirements for the grid index. The reason is that we are dealing with trajectories instead of discrete points. We clip continuous trajectories with all the grid cells that they overlap with. The longer the trajectory the more clippings need to be done, and more grid entries to be made. $T_{objects}$ and $T_{queries}$ represent average number of mini-segments for trajectories (for objects) and trajectory volumes (for queries) during time interval Δ, and "2" stands for 2 entries (points) needed to represent a line. The formulas describing $T_{objects}$ and $T_{queries}$ parameters are below:

$$
\begin{aligned}
T_{objects} &= \left(\frac{\sum_{i=1}^{2+\Delta * r_{obj}} \sqrt{(x_{i+1}-x_i)^2+(y_{i+1}-y_i)^2}}{C_{width}} \right); \\
T_{queries} &= \left(\frac{\sum_{i=1}^{2+\Delta * r_{qry}} \sqrt{(x_{i+1}-x_i)^2+(y_{i+1}-y_i)^2} * H_{ave} * W_{ave}}{C_{width} * C_{height} * C_{length}} \right);
\end{aligned}
\tag{3.4}
$$

We approximate the average length of a trajectory using the sum of distances between 2 points. The

points represent location updates that arrive during time interval Δ. The total length of trajectory is then divided by the size of the grid cell to approximate the number of grid cells that it overlaps with. For queries, we are dealing with volumes, hence we divide the volume of the query during time interval Δ by the volume of the grid-cell.

If M is the total amount of memory available, we can approximate the total number of objects and queries that can be supported by discrete and continuous systems (i.e., approximate the scalability of each system). The total number of queries that can be supported by the discrete system is:

$$N_{qry} = \frac{M - (\Delta * (r_{obj} * E_{obj} + r_{qry} * E_{qry}) + N_{obj} * E_{obj})}{E_{qry} + C_{AR} * E_{grid}}; \tag{3.5}$$

Similarly, the total number of objects that can be supported by the discrete system can be expressed as following:

$$N_{obj} = \frac{M - (\Delta * (r_{obj} * E_{obj} + r_{qry} * E_{qry}) + N_{qry} * (E_{qry} + C_{AR} * E_{grid}))}{E_{obj}}; \tag{3.6}$$

Continuous model can support the following number of queries:

$$N_{qry} = \frac{M - (\Delta * (r_{obj} * E_{obj} + r_{qry} * E_{qry}) + N_{obj} * (E_{obj} + 2 * E_{grid} * T_{objects}))}{E_{qry} + 2 * E_{grid} * T_{queries}}; \tag{3.7}$$

and objects:

$$N_{obj} = \frac{M - (\Delta * (r_{obj} * E_{obj} + r_{qry} * E_{qry}) + N_{qry} * (E_{qry} + 2 * E_{grid} * T_{queries}))}{E_{obj} + 2 * E_{grid} * T_{objects}}; \tag{3.8}$$

In conclusion, we can see that the memory requirements for both discrete and continuous models are the same for all data structures except for the grid index because of different modelling approaches for motion and different representations inside the grid. Once the environment para-

meters (E_{obj},E_{qry}, and E_{grid}) are fixed, the number of objects and queries supported by either of the systems depends on the length of time interval Δ, and rate of arrival of location updates from objects and queries. For discrete model, it also depends on the average size of the queries and the size of the grid cells (i.e., how many grid cells queries overlap with). For continuous model, the length of the trajectories and the size of the grid cells have a significant affect on performance. The longer the trajectories, the more grid cells they are overlapping with, hence more entries need to be made. The lengths of the trajectories can be affected by the following: (1) the speed, (2) the frequency of location updates. If the objects/queries move fast, they cover large distances between the updates. Similarly, if infrequent updates are sent, the objects/queries might cover large distances between the updates as well. In summary, the discrete model has a better performance, hence facilitates scalability (in terms of the number of supported objects and queries) better than the continuous model, but as we will show in the next chapters this comes at the price of accuracy.

Chapter 4

Accuracy

Here I describe my method for estimating accuracy. I use this method in evaluating the quality of results output by discrete model with respect to the continuous model.

4.1 Measuring Accuracy

Since the result formats differ for the discrete and continuous models (the former outputs a discrete point, and the latter outputs a continuous segment), we need to define a method to bridge these two types of results to a common format in order to compare them and approximate accuracy using a numeric value.

We claim that continuous results give a better accuracy than discrete results, and thus better represent the reality. Moreover, continuous answers always contain the discrete answers (i.e., the latter is a subset of the former). We thus compare results of the discrete model to those of the continuous one (i.e., form continuous segments out of the discrete answers and compare them to the equivalent continuous results).

In measuring the accuracy of the results I propose the following approach:

1. For each continuous result in CSA, we determine a length of the answer, where T_{end} and T_{beg} are the ending and beginning timestamps of the result trajectory. We denote it as *continuous result extent* or *CRE*.

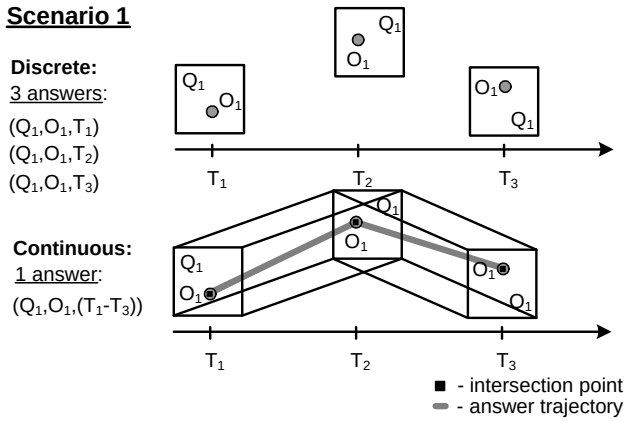


Figure 4.1: Object location update received every time object entered, stayed, and left the query

2. We group the discrete results in DSA by *QID* and *OID*. For each group, we interpolate through discrete results. The key assumption for producing "continuous" results out of discrete answers is not to allow any "gaps". For example, if we got discrete answers with the following timestamps: T_1, T_2, T_3, T_5, T_7, T_8, T_9, T_{10} we calculate only 3 timespans (without any gaps) - [T_1-T3], [T_5], [T_7-T_10]. We denote interpolated points as *discrete result extent* or *DRE*. Notice [T_5] in the example above is called a "lonely" DRE.

3. We compare *DRE*s with *CRE*s:

   (a) We compare the *DRE*s to the corresponding *CRE*s, meaning the timestamps of the discrete results must be within the continuous results.

   (b) For the lonely *DRE*s we introduce a variable λ, that describes a small time interval before and after the discrete result. λ will be calculated for a lonely *DRE*, and will depend on the length of the corresponding *CRE* and the number of location updates received during that time interval.

   (c) We compare lengths and count of *DRE*s to corresponding *CRE*s to determine accuracy.

Let $C = \{CRE\}$ be the set of continuous result extents (i.e., trajectory answers). Then for each $CRE_i \in C$, we find a set of corresponding DRE$_j$s, such that their time intervals overlap. $F(CRE_i) = \{DRE_j \mid DRE_j \cap CRE_i\}$. We compare the sum of lengths of the *DRE*s to the length of corresponding *CRE*.

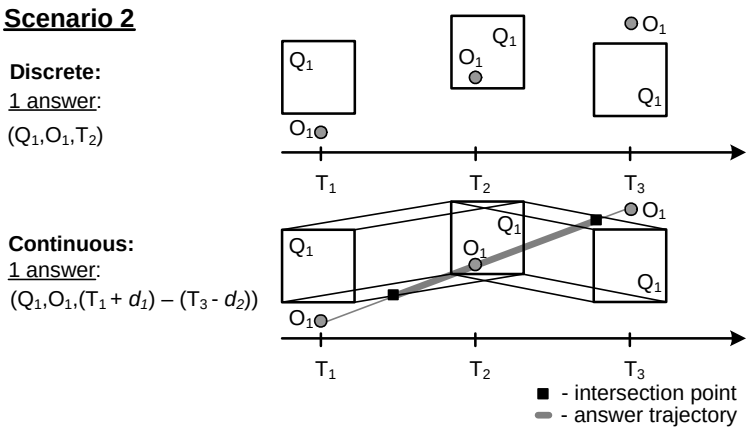


Figure 4.2: Object location received only once when object was inside the query

$$Accuracy = \sum_{i=0}^{n} \frac{\sum_{j=CRE_i.t_{start}}^{CRE_i.t_{end}} length(DRE_j)}{length(CRE_i)} \tag{4.1}$$

Scenarios that illustrate my approach in measuring and comparing accuracy of the two models are:

Scenario 1: *Object sent location updates when it entered the query, was inside the query, and left the query*. In this scenario, location updates for both objects and queries are known for every time unit when the object was inside the query. Figure 4.1 illustrates this case. This is the best case for DSA. Every time unit we know the positions all objects and all queries. Thus we can accurately determine if an object is inside a query at every time unit. Using my accuracy model, we find that

$$\textit{length}(CRT_1) = (T_3 - T_1) = (3 - 1) = 2;$$

and

$$\textit{length}(DRE_1) = [T_3 - T_1] = 3 - 1 = 2.$$

Then using equation (4.1).

$$\textit{Accuracy} = \textit{length}(DRE_1)/\textit{length}(CRT_1) = 2/2 = 1(\sim 100\ accuracy)$$

By my model, results returned by the DSA are as accurate as the results returned by CSA. This

scenario is highly unlikely in the real life situation when given a large number of queries, due to the fact that it is hard to guarantee that location updates for every time unit object was inside a query are received. During the execution, the data streams can become bursty, and some tuples might have to be dropped. Due to network congestions some updates might arrive late. Devices can be configured differently to send their location updates. Therefore, in most cases, it is hard to have a setup guaranteeing the best case accuracy for the discrete model.

Scenario 2: *Object sent one location update when it was inside the query*. This is a slightly worse example for the discrete system. Figure 4.2 illustrates this case. We got explicit location updates at T_2 for both an object and a query. So we have one discrete and one continuous answer. To compare the accuracy using the approach above, we calculate the following:

$$\textit{length}(CRT) = ((T_3 - d_1) - (T_1 + d_2))$$

where d_1 and d_2 are distance measurements from the discrete time units to indicate that the intersections occurred not exactly at T_1 and T_3. For this example, we assume they are something arbitrarily small, but in real experiments we can calculate these values exactly. DRE = [T_2] (i.e., it's a *lonely* DRE, so we need to determine a continuous interval for it). To set an appropriate value, we divide the *length*(CRT) by the (number of location updates received - 1) to get the average length of a line segment ($S_{ave-length}$).

$$S_{ave-length} = \textit{length}(CRT)/(\#updates - 1)$$

So in this case, $S_{ave-length}$ = 2/(3-1) = 1. Then DRE = [T_2] = 1*$A_{discrete}$, where $A_{discrete}$ is the number of discrete answers. Then Accuracy = (1)/((3-0.02)-(1+0.01)) = (1)/(1.97) = 0.507 ($\sim 50\%$ accurate). This value is an approximation of accuracy, since we picked arbitrary values for d_1 and d_2.

Scenario 3: *No location update was received while object was inside the query*. This is the worst-case scenario example for the discrete system. The server didn't receive a single location

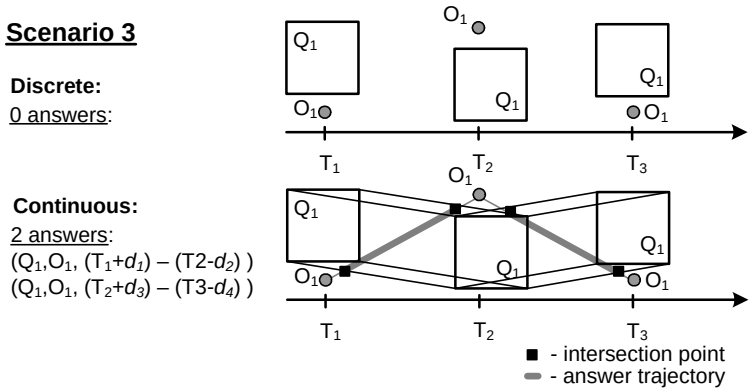


Figure 4.3: No location update received at any point in time when object was inside the query

update from the object while it was inside the query. Using the discrete model, we would not get an answer that the object was inside the query. Figure 4.3 illustrates this case. Using my measuring approach, we can determine that *Accuracy* = 0 %, due to 0 answers returned by the discrete model. So the *length*(DRE) = 0, and the discrete model is 0% accurate.

I tested my model on a number of experiments to support my initial intuition with respect to accuracy of the two models.

4.1.1 Analysis of Accuracy

Below I analyze accuracy using an example of one moving object and one query. I consider the parameters listed in Table 4.1 for this analysis.

Variable	Description
Δ	Time interval between periodic query execution
d_{delta}	Average distance travelled between executions
d_{update}	Average distance travelled between updates
d_{join}	Average distance object is inside the query
d_{weight}	Average weight of each update in the overall answer
v	Average velocity of a moving entity
u	Average number of location updates per Δ
u_{join}	Updates contributing to *intersection* w/ query
$a_{discrete}$	Number of discrete results

Table 4.1: Parameters for accuracy comparison

An average distance travelled by a moving object between the times when joining with the

query can be expressed as

$$d_{delta} = \Delta * v \tag{4.2}$$

An average distance travelled between location updates is

$$d_{update} = \frac{\Delta * v}{u + 1} \tag{4.3}$$

An average length (trajectory) when moving object is inside the query is

$$d_{join} = d_{delta} * \varphi \tag{4.4}$$

where φ is an intersection factor (between 0 and 1). This factor describes the probability of an object and query to be *joinable* (i.e., producing an answer). It is affected by the type of the movement of the object and the query, their velocities, network constraints, and the type of the query. Note, we assume that the greater the velocity, the greater is the φ join factor. When φ = 1, it means that an object is inside the query (i.e., produces an answer for the entire duration of time interval Δ).

By dividing d_{join} by the number of updates that contributed to the continuous answer u_{join}, we get an average continuous interval segment. In other words, we approximate an average mini-segment of the continuous answer per each point that contributed to the overall continuous answer.

$$d_{weight} = \frac{d_{join}}{u_{join}} \tag{4.5}$$

The discrete result extent (*DRE*) can then be determined as

$$DRE = d_{weight} * a_{discrete} \tag{4.6}$$

Combining (4.2), (4.4) and (4.5), *DRE* can be expressed as

$$DRE = \frac{\Delta * v * \varphi}{u_{join}} * a_{discrete} \tag{4.7}$$

We compare accuracy of a discrete model by dividing the discrete result extent by a continuous

result extent, which can be expressed as

$$Accuracy = \frac{DRE}{CRE} = \frac{\frac{\Delta * v * \varphi}{u_{join}} * a_{discrete}}{\Delta * v * \varphi} = \frac{a_{discrete}}{u_{join}} \tag{4.8}$$

Equation (4.8) confirms my intuition that becomes evident through the experimental studies. The accuracy of the discrete model would be the ratio of the discrete answers (which correspond to the explicit location updates) to the number of the discrete updates out of which the continuous model was able to *calculate* the answers.

If the discrete model doesn't receive the location update when the object is inside a query (e.g., range query) discrete model would not be able to return such an answer. This becomes a problem when objects move extremely fast or location updates are sent infrequently, or not every location update is received (e.g., network delay, packet loss, load shedding, etc).

Chapter 5

Experimental Results: Accuracy vs. Performance Tradeoff

Here I assess the performance and accuracy of DSA and CSA in evaluating a set of continuous spatio-temporal range queries.

5.1 Experimental Settings

I have built motion operators implementing DSA and CSA algorithms within our stream processing system CAPE [64] to evaluate their performance and accuracy. Moving objects and moving queries generated by the *Network-Based Generator of Moving Objects* [10] are used as data. The input to the generator is the road map of Worcester, USA (where Worcester Polytechnic Institute is located).

All the experiments were performed on Red Hat Linux (3.2.3-24) with Intel(R) XEON(TM) CPU 2.40GHz and 2GB RAM. The set of objects consists of 5,000 objects and the set of queries consists of 5,000 continuous spatio-temporal range queries. To get the most accurate comparison between the two models, the number of objects and queries was kept constant (i.e., no new objects or queries enter the system, and no existing objects and queries disappear). Each moving object or query reports its new information every time unit, unless explicitly specified otherwise. Our periodic execution was set to 2 time units. Thus 2 location updates were received per object or per

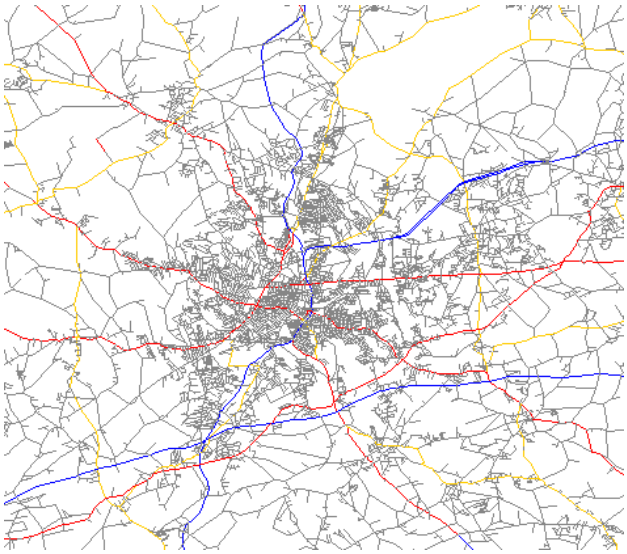


Figure 5.1: Road network map of Worcester, MA

query during each evaluation interval.

I ran experiments varying the speed and the update probabilities of objects and queries observing their effect on the performance and the accuracy of the two models.

5.2 Varying Speed

Here, I vary the speed of objects and queries. The speed of the moving objects and queries is defined by the *max.speed div.* parameter specified to the data generator. The larger this value, the slower the moving objects. The parameter was varied from 1 (very fast) to 250 (very slow). Figure 5.2 gives the effect of increasing the speed of objects and queries on the join time[1]. When objects move slowly and regularly sent their location updates, both models exhibit similar performance (Figure 5.2). The join time for the discrete model stays relatively constant, where as for the continuous mode, it goes up as the speed increases (e.g., speed = *fast* or *very fast*). The reason is that objects cover large distances between their location updates, for which the continuous model constructs trajectories and then clips those to grid cells. These mini-segments of trajectories (for moving objects) and trajectory volumes (for queries) might overlap with a large number of grid

[1]Time is measured using wall clock time.

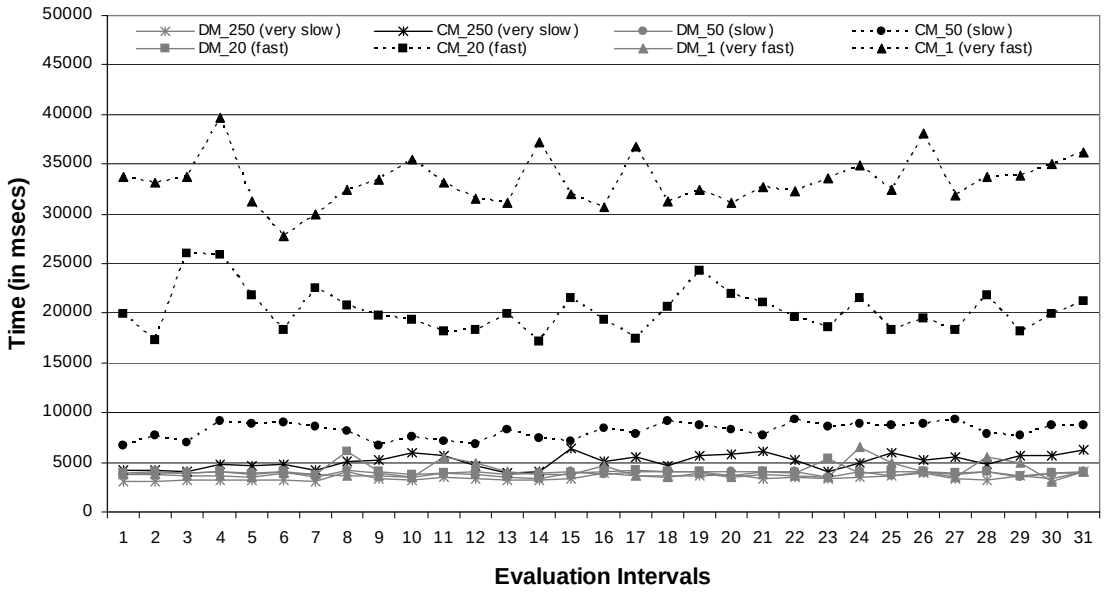


Figure 5.2: Performance of discrete and continuous models with varying speed of moving objects and queries

cells causing a lot of comparisons between objects and queries. The speed of moving objects and queries has no effect on the discrete model performance. What happened in between updates (i.e., the distance covered by objects and queries) is of no concern to the discrete model.

Figure 5.3 shows the average accuracy for both models when the speed of objects and queries is varied. Accuracy has been measured for the results output by the discrete model with respect to continuous model results using our accuracy model in Section 5.1. When the speed is slow, the accuracy of the discrete model is close to the continuous model (e.g., *very slow* $\approx$ 97.08% and *slow* $\approx$ 90.62%). The reason for that is that when objects and queries move slowly, they cover small distances between their location updates. Thus, there is only a rather small chance that an intersection between object and a query occurs and no location update is sent. On the opposite, when objects move very fast, and objects and queries cover large distances between the updates, discrete model would not output the query-objects intersections that might have occurred between the updates.

Figure 5.4 illustrates the tradeoff between the two models in terms of join time and accuracy. For slower moving objects, discrete model has a pretty high accuracy with a much better performance than the continuous model. But as the speed of both objects and queries increases, the

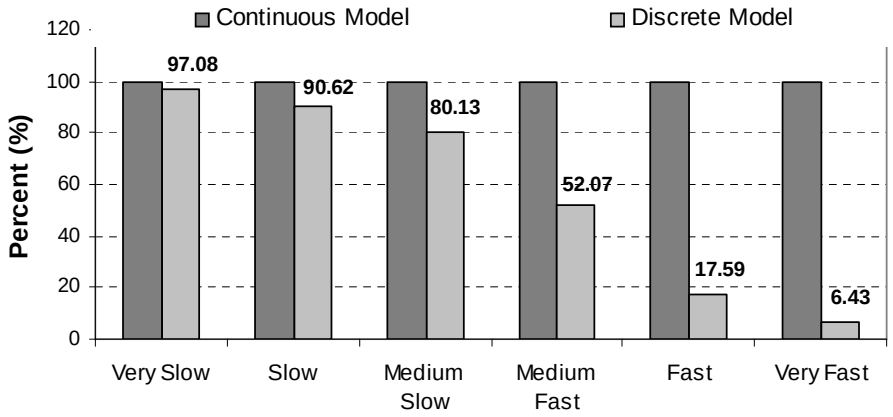


Figure 5.3: Accuracy of discrete vs. continuous models with varying speed of moving objects and queries

accuracy of the discrete model drops significantly (down to approximately 6.43%) when objects move extremely fast. The high accuracy of the continuous model comes at a price of a much more expensive join time, approximately 7 times slower.

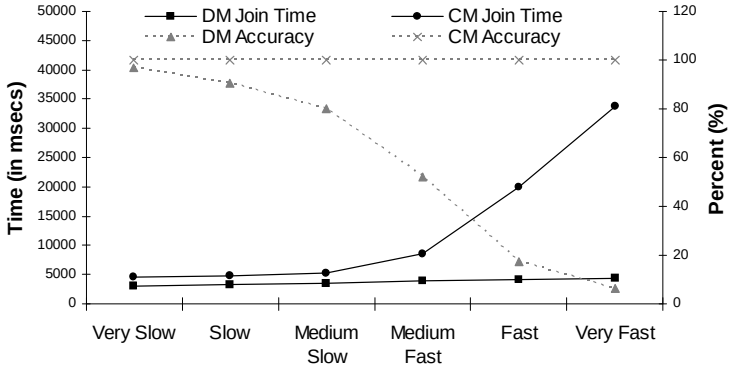


Figure 5.4: Performance vs. accuracy when varying the speed

5.3 Update Probability

In this experiment, I evaluate the accuracy of the discrete and continuous models under various update probabilities[2] of objects and queries. This experiment is designed to help us to see how much missing information (or load shedding) can be afforded without any or very little loss in accuracy. I ran experiments on the continuous model, varying the update probability of both moving objects

[2]Defines the probability of reporting a moving object. 1000 means that a moving object is reported at every time stamp during its move. 500, e.g., means that an object is reported with a probability of 50%.

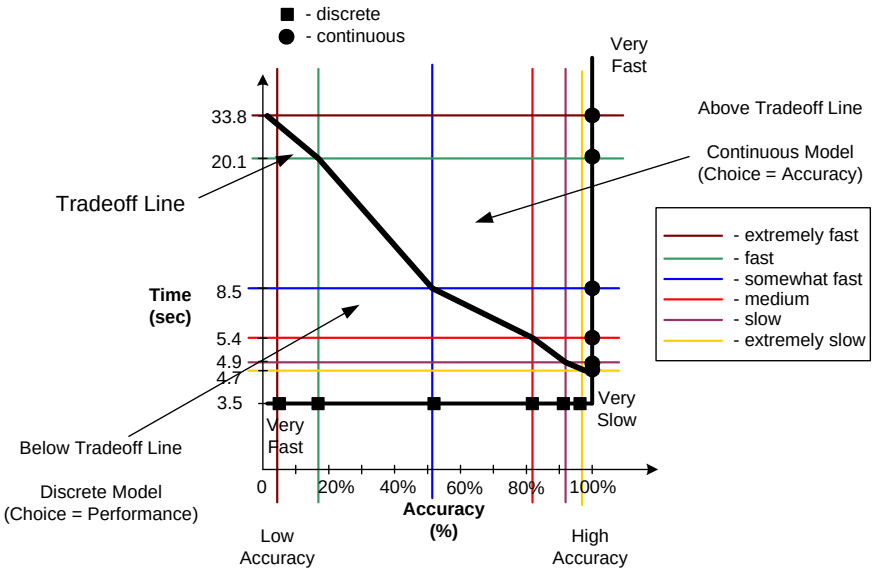


Figure 5.5: Tradeoff between performance and accuracy when varying the speed (Version 2)

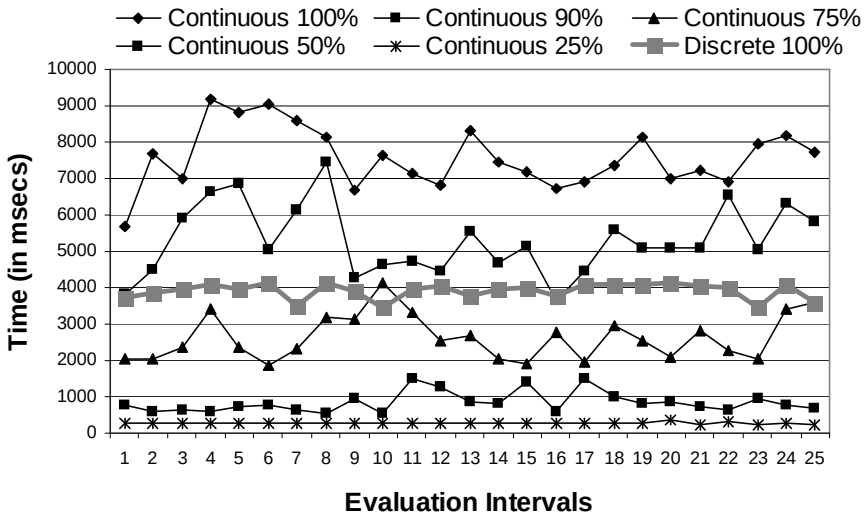


Figure 5.6: Performance of continuous model with varying update probabilities

and queries, keeping the speed parameter constant at a medium speed (speed parameter = 50). The probability of reporting a moving object and query was varied from 100% to 25 %.

Figure 5.6 shows the join times for the continuous model with different update rates. The smaller the update rate, the fewer location updates need to be processed per evaluation interval also the fewer computations need to be made making the join cheaper. Figure 5.7 shows the accuracy of the continuous model with varying update rates. The results output by the continuous model with varied update probabilities were compared to the results of the continuous model when

100% of the location updates were received. I included discrete model (performance and accuracy) for comparison to see how the discrete model with 100% of location updates compares with the continuous model with smaller update probabilities. With 75% of updates from both objects and queries at a medium speed, continuous model gives a higher accuracy ($\approx$ 68.5 %) than the discrete model ($\approx$ 62.1 %) and better performance ($\approx$ 2635 ms for continuous vs. 3913 ms for discrete). Figure 5.8 graphically depicts the tradeoff between the performance and accuracy when update probabilities of moving objects and queries vary. Figure 5.5 portrays a tradeoff between the performance and accuracy in a different fashion. Here there two nested axis. The outer axis (with the origin in the lower left corner) has x-axis representing accuracy in terms of a percentage value, and y-axis representing execution (join processing) time. The nested axis (with the origin in the lower right corner) has x-axis (going from right to left) representing the discrete model and y-axis (going from bottom to top) representing the continuous model when varying the speed of the objects and queries. At the origin, the objects are moving extremely slow, and the further the values are from the origin of the nested axis the faster the speed is. The thick line represents a tradeoff line between the performance and the speed for both discrete and continuous models. Above the tradeoff line, the weight is put more on the accuracy, hence the continuous model would be a better choice. Below the tradeoff line, the emphasis is on the performance, thus the discrete model would suffice.

5.4 Analysis of Affecting Factors

Velocity and update probability affect performance and accuracy of the discrete and continuous models. The continuous model is more preferred when: (1) objects move fast; (2) not all location updates are received (e.g., load shedding occurs); (3) location updates arrive out-of-sync due to network delay (in this case, we assume the system would load shed this data, as it is outside of the current window of execution). Continuous model can approximate where objects and queries are even if no explicit location update is received. Continuous model can give a higher accuracy with

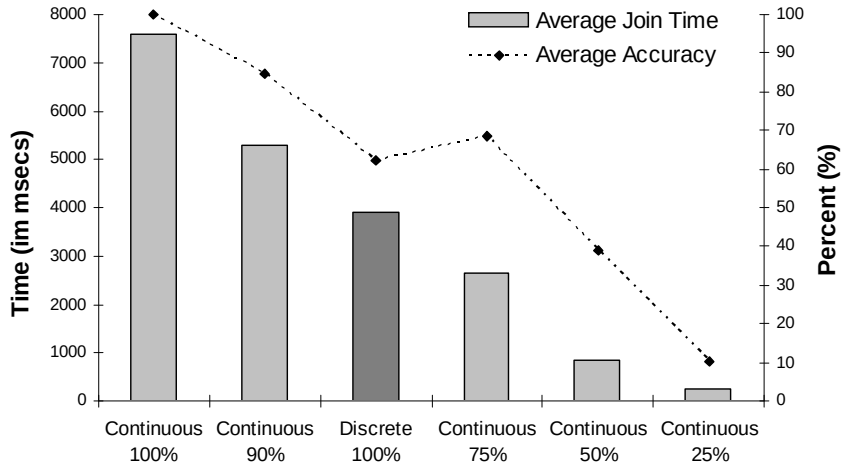


Figure 5.7: Accuracy of continuous model with varying update probabilities vs. discrete model with 100% update probability

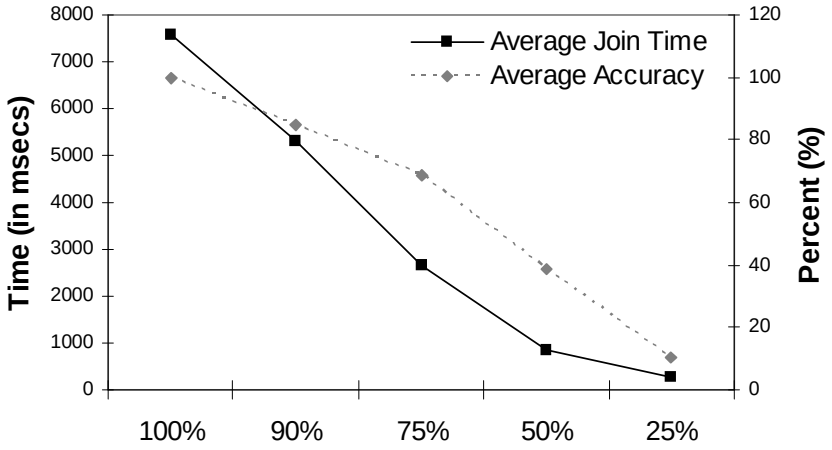


Figure 5.8: Tradeoff between performance and accuracy when varying the update probability

better performance with only 75% of location updates.

Discrete model suffers when objects move fast and rarely send their location updates. The distance they cover between the updates could be large and the discrete model has no way to approximate where objects are in-between the updates. Hence, discrete model is preferred when objects move slow, or when very frequent location updates occur.

Chapter 6

Part III:

Scalable Cluster-Based Execution

In this part of the thesis, I propose clustering spatio-temporal data streams based on common attributes in order to improve performance when evaluating spatio-temporal queries. In addition, I propose moving cluster-based load shedding to reduce system overload while preserving a relatively good quality answers for spatio-temporal queries.

6.1 Choosing a Clustering Algorithm

Before I proceed with the details of the proposed cluster-based execution (Chapter 7), I describe my choice of clustering algorithm and support it by analyzing and comparing it to other clustering algorithms in terms of memory, performance and robustness.

6.2 Clustering Basics

There are many definitions of cluster analysis in the literature. The following is from [40]:

The goal of cluster analysis is to partition the observations into groups (clusters) so

that the pair wise dissimilarities between those assigned to the same cluster tend to be smaller than those in different clusters.

It is important to understand the difference between clustering (*unsupervised classification*) and discriminant analysis (*supervised classification*) [47]. In supervised classification, we are provided with a collection of *labeled* (pre-classified) patterns; the problem is to label, a newly encountered, yet unlabeled, data. An example of supervised classification could be, for example, a grid index, where moving objects are hashed based on their locations to pre-defined buckets (pre-determined by the grid cells' size and count). In the case of clustering, the problem is to group a given collection of unlabeled data points into meaningful clusters. In a sense, clusters are unknown a priori and thus are solely data-driven.

6.2.1 K-Means Clustering Algorithm

K-means is one of the most common clustering algorithms. It is an iterative clustering method that takes quantitative data as input and measures the similarity among the various data points by calculating a distance measure, typically the squared Euclidean distance (Equation (6.1)):

$$|a-b| = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2} = \sqrt{(a_1-b_1)^2 + ... + (a_n - b_n)^2} \tag{6.1}$$

where a and b are points in the Euclidean space $\Re^n$. Although there are many ways to calculate the similarity between data points, I use the squared Euclidean distance measure in this work. K-means is a static clustering method. It is done once all data available at a specific point in time.

K-means consists of two phases: First, all data points are assigned to the closest cluster. Second, the algorithm determines the cluster means. K-means repeats these two steps until it converges. Because the similarity between data points is measured using Euclidean distance, the K-means algorithm is sensitive to outliers and tends to recognize spherical patterns [31]. The criterion function usually keeps track of the total sum of all distances between a data point and its closest cluster. If changes in this function are considered marginal, K-means terminates and is said

to have converged [63]. Although K-means converges (i.e., it stops after a finite number of steps), the number of steps is not known in advance.

In [21], the complexity for K-means is described as *O(n*d*k*iter)*, where *d* is the number of features, *iter* the necessary iterations, which are usually much less than the number of data points, *n* is the number of data points and *k* is the number of clusters.

6.3 Approaches To Incremental Clustering

When picking a clustering algorithm, I considered several criteria. Table 6.1 shows a list of criteria from the literature for incremental clustering algorithms. In particular, I was concerned whether it would be more advantageous in terms of performance and accuracy to keep the number of clusters or the threshold parameter constant.

6.3.1 Criteria for Scalable Clustering

According to the suggestions found in the literature, an incremental algorithm should meet the following criteria listed in Table 6.1.

For SCUBA, I considered clustering algorithms based on the following four criteria: *Performance*, *Memory*, *On-line Clustering Capabilities* and *Robustness*. My requirements for clustering algorithms are listed in Table 6.2.

6.3.2 Determination of Parameters

The number of clusters has a direct effect on the performance and memory requirements of a clustering algorithm. The total number of clusters of a model is either determined by *k* (number of clusters), by Θ (threshold) or by η (learning rate). I am going to discuss in this thesis only *k* and Θ. Θ is a threshold that triggers the creation of a new cluster. A low threshold makes the creation of a new cluster more likely than a high value of Θ and Θ can be used to limit the number of clusters [21, 15]. Because an algorithm, such as Leader-Follower, creates a new cluster for every

Source	Requirement	Criteria
Domingos et. al [20]	Must require small constant time per record, otherwise it will fall behind the data.	*Performance*
	Must use only a fixed amount of main memory, irrespective of the total number of records it has seen.	*Memory*
	Must be able to build a model using at most one scan of the data, since it may not have time to revisit old records, and the data may not even all be available in secondary storage at a future point in time.	*On-Line Clustering Capabilities*
	Must make a usable clustering model available at any point in time, as opposed to only when it is done processing the data, since it may never be done processing.	*On-Line Clustering Capabilities*
	Should produce a model that is equivalent (or nearly identical) to the one that would be obtained by the corresponding ordinary algorithm, operating without the constraints found in streaming environments.	*On-Line Clustering Capabilities*
	When data is changing over time, the model at any time should be up-to-date, and also include all information from the past that has not become outdated.	*On-Line Clustering Capabilities*
Ye et. al [86]	Classification precision	*On-Line Clustering Capabilities*
	Scalability of learning and classification on large data sets	*Performance*
	Robustness to noise	*Robustness*
	Ability for incremental learning	*On-Line Clustering Capabilities*
Barbara et. al [6]	Compactness of representation	*Memory*
	Fast, incremental processing of new data points	*Performance*
	Clear and fast identification of outliers	*Robustness*
Gupta et. al [30]	Given n data points, algorithms should have O(n) time complexity and O(1) space complexity.	*Performance*

Table 6.1: Requirements for incremental data stream analysis algorithms [43].

data point, if the distance to the closest cluster exceeds Θ, a small Θ makes it more likely that the online clustering algorithm creates a new cluster and overall the number of clusters gets very large.

6.4 Leader-Follower Clustering Algorithm

The *Leader-Follower* (LF) clustering algorithm is a simple incremental clustering algorithm. It allows the number of clusters to grow until the boundary is reached. The maximum number of

Requirement	Criteria
Only the cluster centers need to be stored, data points can be discarded after clustering.	*Memory, On-Line Clustering Capabilities*
Runs in constant time.	*Performance*
Number of clusters can be controlled or varied.	*Memory*
Parameters can be easily determined.	*Ease of use*
Number of parameters is small.	*Ease of use*
Robust for highly dynamic spatio-temporal data sets.	*Robustness*

Table 6.2: My criteria for online clustering algorithms

clusters is *k* = *n*. The threshold Θ determines the number of clusters. A large threshold means that clusters will contain objects that have a larger dissimilarity, a small threshold, on the other hand, will increase the probability that the LF algorithm creates a new cluster. The similarity between data points is measured using the Euclidean distance.

The pseudo-code for the Leader-Follower Clustering Algorithm is shown below (Algorithm 4).

Algorithm 4 *Leader-Follower()*

initialize η, Θ
$m \leftarrow x_1$
repeat
  accept new x
  $j \leftarrow \arg\min_j \|x - m_j\|$ *//find nearest cluster*
  if $\|x - m_j\| > \Theta$ **then**
    create new cluster $m \leftarrow x$
  else
    m_j = (1 - η)m$_j$ + η x
  end if
until no more data points
return m_1, m_1,...,m_k

How does LF come up with an acceptable solution regarding the quality of the clustering? LF optimizes a criterion function, namely the average squared distance from data points to the closest cluster, by always choosing the closest cluster for a new point. This is also the case in the first phase of the K-means algorithm [63]. In addition to that, K-means also computes the means of all clusters based on the associated data points. By doing this iteratively, it finally ends up with a

minimized criterion function. Although LF does not iterate over the data set over and over again, it does minimize the error function in that choosing any other cluster than the closest for a given data point would result in a higher distance.

The quality of K-means is superior to the quality of LF but it comes with more computational effort. K-means iteratively minimizes the total sum of distances between all data points and their associated closest cluster. This explains that K-means requires higher computational efforts (Table 6.1). LF, on the other hand, minimizes the total sum of all distances by always confining a new data point to the closest cluster. One can expect that LF would be much faster for a large number of clusters and not much faster for smaller number of clusters, if compared to K-means. The reason is because K-means requires less iterations to converge if *k* is small.

The number of clusters has a linear influence on both algorithms, due to the calculations of the distances between all the clusters and a new data point. The high variance of K-means results from the random initialization of the cluster means. This in turn has an impact on whether the algorithm converges early. Because the number of scans over one data set of the LF is always one, the complexity is at least *O(n*k)*.

So when choosing between K-means and LF, one is faced with a trade-off between performance and quality. In this thesis, I chose the LF algorithm. As experiments will later show, the quality of clusters that LF algorithm produces is relatively good with a much better performance comparing to the K-means algorithm.

6.4.1 Analysis of the Leader-Follower Algorithm

Below I analyze the LF algorithm based on the criteria in Table 6.2 in Section 6.3.1.

Criterion 1: On-line Clustering Capabilities . The LF algorithm in its basic form incrementally clusters data.

Criterion 2: Memory . The memory requirements are *O(k)*. The dimensions of the data and the number of clusters define the space requirements. High memory requirements result from

the increasing number of clusters, each of which needs a chunk of space to store the cluster centers.

Criterion 3: Performance . The complexity of the LF algorithm is *O(k*n)*, where *k* is the number of clusters and *n* is the number of data points. The reason for such complexity is because it performs only one scan over data points that have to be processed and the number of clusters has an impact on calculations to determine the closest cluster center of a data point.

Criterion 4: Robustness . If not handled explicitly, outliers are likely to lead to a new cluster. This in turn may lead to a lot of single-member clusters. In its basic form, the algorithm does not provide any tools to handle outliers and thus cannot be considered robust in that sense.

6.5 Competitive Learning Clustering Algorithm

Another clustering algorithm candidate I considered was the *Competitive Learning* (CL) algorithm. Contrary to the LF, CL algorithm controls the number of clusters. The parameter *k* is hard-coded in the design of the algorithm.

The number of clusters is constant and is a part of the design decision. The basic concept of CL is that changes affecting the present clustering are confined to exactly one cluster (i.e., one that is closest to a new data point). This is contrary to K-means, where adding a data point could lead to changes to the overall cluster structure, because cluster means are iteratively re-assigned until the algorithm converges. The Competitive Learning algorithm originated from neural networks research [9, 41]. The pseudo code of CL is shown below (Algorithm 5).

The whole data set is presented to the algorithm several times, until convergence is reached. The data set is shuffled each time before it is processed again.

Because of calculating the scalar product of cluster weights and data point, only their respective angle is relevant to determine the closest cluster to data point. The absolute size of a cluster cannot dominate over other clusters of the cluster structure. Further, only a cluster affected by adding a

Algorithm 5 *CompetitiveLearning()*

initialize η, Θ, $iter$, $w_1, ...w_k$
repeat
  $x_i \leftarrow \{1,x_i\}$, i=1,...,n *//augment all patterns*
  $x_i \leftarrow x_i$ / $\|x_i\|$, i=1,...,n *//normalize all patterns*
  $j \leftarrow \arg\max_j \left\|w_j^T, x\right\|$ *//classify x*
  $w_j \leftarrow w_j + \eta x$ //weight update
  $w_j \leftarrow w_j$ / $\|w_j\|$, j=1,...,k *//normalize weights*
until no significant change in w in $iter$ attempts
return $w_1...w_k$

data point is altered. Similar to K-means, CL converges at a user-defined level, optimizing some squared-error function.

Because this algorithm runs continuously until convergence is reached (or a maximal number of *iter* iterations) and no guarantee exists that this condition can be reached, a decay rate that affects the learning rate has to be specified to prevent the algorithm from alternating weights forever. Such a decay rate can also have a negative impact. It might become so small that new data points will not be learned. More details about this algorithm can be found in [21], and information about competitive learning in general is available in [41].

6.6 Analysis of the Competitive Learning Algorithm

Below I analyze the Competitive Learning Algorithm based on the criteria in Table 6.2.

Criterion 1: On-line Clustering Capabilities . Although this algorithm needs several scans over the data, it can be implemented as a single-scan algorithm, i.e., data can be clustered with one scan. Doing so makes it more or less a *Leader-Follower* algorithm with a fixed number of clusters. The initial cluster centers would be initialized in a way as to set their values to an initially calculated cluster structure. Depending on the time that is available to process the new data set, allowing for some iterations to adapt to the new clusters could improve the accuracy of a clustering.

Criterion 2: Memory . A compact representation of the cluster structure is possible, because only the cluster weights are kept in memory, if the CL algorithm is implemented as a single-scan algorithm. If the algorithm is implemented in a way where several scans over the data are allowed, memory requirements will depend on the size of the data set and the number of clusters that will be stored in memory. The memory requirements of the single-scan algorithm are *O(k)*, for more than one scan it is *O(k+n)*, where *n* denotes the number of data points and *k* the number of clusters.

Criterion 3: Performance . As a single-scan algorithm, the computational complexity is *O(n)*. Otherwise, it becomes *O(n*iter)*, where *n* denotes in both cases the number of data points and *iter* the number of iterations until convergence is reached.

Criterion 4: Robustness . Like the K-means, CL is sensitive to outliers and offers no method to prevent the algorithm from incorporating such data points into the cluster structure. In the case of the single-scan implementation, the outcome is expected to be worse than K-means, because it only partly maximizes a criterion function.

6.7 Comparing Competitive Learning, Leader-Follower and K-Means Algorithms

The following table summarizes the comparison of the K-means, Leader-Follower, and Competitive Learning Clustering algorithms.

6.7.1 Clustering Algorithms Comparison Summary and My Choice

The K-means, Leader-Follower and Competitive Learning algorithms solve the online clustering problem from different perspectives: either with a constant number of clusters *k* or the threshold distance Θ. While CL is not an online clustering algorithm, it can be modified such that it allows clustering data sets with one single scan over the data set. The fact that the number of clusters

Criterion	K-means	LF	CL
On-line Clustering Capabilities	Some windowed versions of K-means exist, which allow processing data points incrementally. We note that if K-means is made an incremental clustering algorithm, it becomes very similar to LF.	Generic model for an incremental clustering algorithm. Can be extended according to several suggestions and ideas in the literature [43, 39]. The number of clusters is controlled by parameter Θ.	Can be achieved if the algorithm is prevented from iterating. Static with respect to the number of clusters, but the quality of the clusters might be poor.
Memory	K-means stores data points and cluster centers.	Only the cluster means are stored in memory.	Only the cluster means are stored in memory. If the algorithm is to iterate over the data points, additional memory has to be allocated.
Performance	*O(n*k*d*iter)* Because the initialization of the clusters is random, the algorithm has an unpredictable, nondeterministic run time. To limit the number of iterations to prevent the algorithm from running until convergence is reached, *iter* must be set accordingly.	*O(n*k)* Runs in predictable time. Outperforms K-means especially for a high *k*.	*O(n*k)* for one scan over the data. Fast algorithm as it performs only inexpensive computations such as calculating the distance between clusters by means of the *cos(x)* function.
Robustness	Cannot handle outliers.	As outliers may lead to new clusters, nonsignificant clusters can be deleted from the model according to a predefined threshold.	Cannot handle outliers.

Table 6.3: Qualitative comparison of the K-means, Leader-Follower and Competitive Learning algorithms

must be defined in advance is considered a major disadvantage, as it makes this algorithm too static compared to Leader-Follower. Thus my preferred choice is the *Leader-Follower* clustering algorithm.

LF is a generic algorithm for online clustering problems. It is considered superior to CL. Its

performance, which is *O(n*k)*, is less expensive computationally than K-means. In addition, it handles outliers better than either the K-means or Competitive Learning algorithms.

Chapter 7

Scalable Cluster-Based Algorithm for Evaluating Continuous Spatio-Temporal Queries on Moving Objects (SCUBA)

In this chapter, I describe, SCUBA, a Scalable Cluster Based Algorithm for evaluating a large set of continuous queries over spatio-temporal data streams. The key idea of SCUBA is to group moving objects and queries based on common dynamic properties (e.g., speed, destination, and road network location) at run-time into moving clusters to reduce data size, improve performance and facilitate scalability. SCUBA exploits shared cluster-based execution by abstracting the evaluation of a set of spatio-temporal queries as a spatial join first *between* moving clusters and then *within* moving clusters. If the clusters don't satisfy the join predicate (i.e., don't overlap), the objects and queries that belong to those clusters can be discarded being guaranteed to not join individually either. This provides cost savings and speeds up the processing. I describe the details of SCUBA algorithm in the sections below.

7.1 Why Current Solutions Might Not Be Adequate

As it was mentioned in the introduction, scalability is a critical concern, when faced with an extremely large number of concurrent queries. In particular, the goal is to reduce memory requirements and to speed up processing. So far we used a shared execution paradigm, where we group moving objects into Objects Table and queries into Queries Table, and then perform a grid-based join between moving objects and the queries based on their locations. There are still several problems with this model:

- When performing a join, we process each moving object and moving query individually. With an extremely large number of objects and queries lot's of comparisons must be made. This can create a bottleneck in performance.
- We store all location updates for objects and queries individually, which potentially can cause the system to run out of memory.
- Given limited memory resources, we might not be able to support a really large number of objects and queries (i.e., can't store all location updates for all moving entities).

There are several possible solutions to the problems listed above. We can perform load shedding to minimize memory usage and speed up the processing. Unfortunately, this is an extreme solution and should be the last resort when the system cannot process the data and is threatened to run out of memory or crush.

An alternative solution could be to set a limit on how many objects and queries a system can support, and reject all new objects and queries once the limit is reached. This solution is also not adequate. Limiting the system to a certain number of objects (users), we ignore all potential users that would like to use our system due to system capabilities limitations. In a real world this could mean a loss of clients, and business. Such small scale solution might not satisfy many applications.

A third solution could be to use distribution. But distribution has its own disadvantages as well. In particular, network delays can lead to obsolete data, and return either incorrect out of date

Figure 7.1: Motivating examples for moving clustering

results or no results at all. Distribution adds more complexity compared to a centralized system. This increases manageability - more effort is required for system management in addition to real-time processing. Because multiple machines are involved the responses can be unpredictable, depending on system organization and network load. Lastly, distributed systems can be more expensive than centralized systems.

I propose an alternative method, the *Scalable Cluster-Based Algorithm* (SCUBA, for short) for evaluating continuous spatio-temporal queries on moving objects. SCUBA exploits the *shared cluster-based execution* paradigm to optimize the execution. By utilizing the shared clustering, moving objects and queries are grouped together into clusters based on common spatio-temporal attributes and then the execution of queries is abstracted as a *join-between* clusters and *join-within* clusters. By using clusters, we also achieve data compression, saving in memory and making the evaluation of continuous queries more efficient.

Unlike most of the previous works on shared execution as means to achieve scalability, where objects and queries are grouped separately; SCUBA focuses on shared execution for heterogeneous entities. Namely, both moving objects and queries grouped into clusters based on similar spatio-

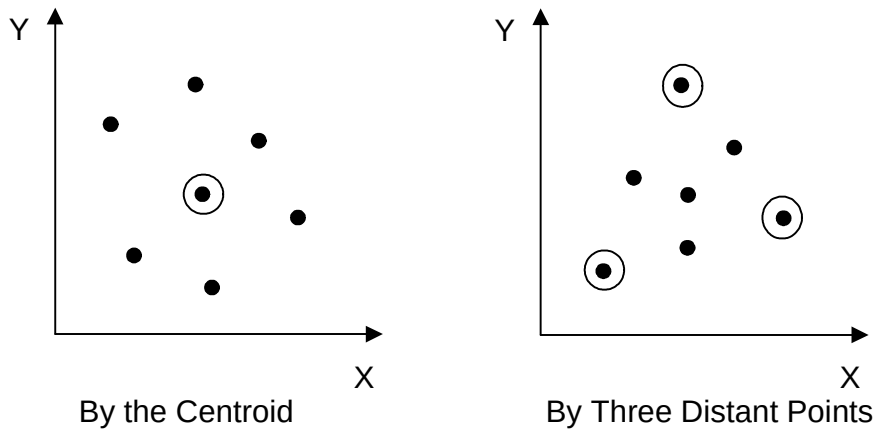


Figure 7.2: Representation of a cluster

temporal attributes. These moving clusters serve as means to improve the performance and achieve scalability. For simplicity, I present SCUBA in the context of continuous spatio-temporal range queries. However, SCUBA is applicable to a broad class of spatio-temporal queries (e.g., *knn* queries, trajectory and aggregate queries).

SCUBA introduces a general technique for processing a large number of simultaneous spatio-temporal queries. Similar to SINA [57] and SEA-CNN [84], SCUBA is applicable to all mutability combinations of objects and queries: (1) Stationary queries issued on moving objects. (2) Moving queries issued on stationary objects. (3) Moving queries issued on moving objects.

When an object or a query belongs to a cluster, the individual location updates for this object/query no longer need to be processed individually. This provides a great amount of memory/storage savings and improvement in processing time as the processing is first done at the higher level of abstraction(level of moving clusters). If the clusters don't satisfy a join condition, there is no need to join individually objects and queries that belong to that particular cluster. Once clusters are formed, they can be treated just as regular moving objects. Thus all of the existing algorithms and indexing techniques can be easily extended to moving clusters. Although, I don't explore this in this thesis, I believe, it can be done without any significant problems.

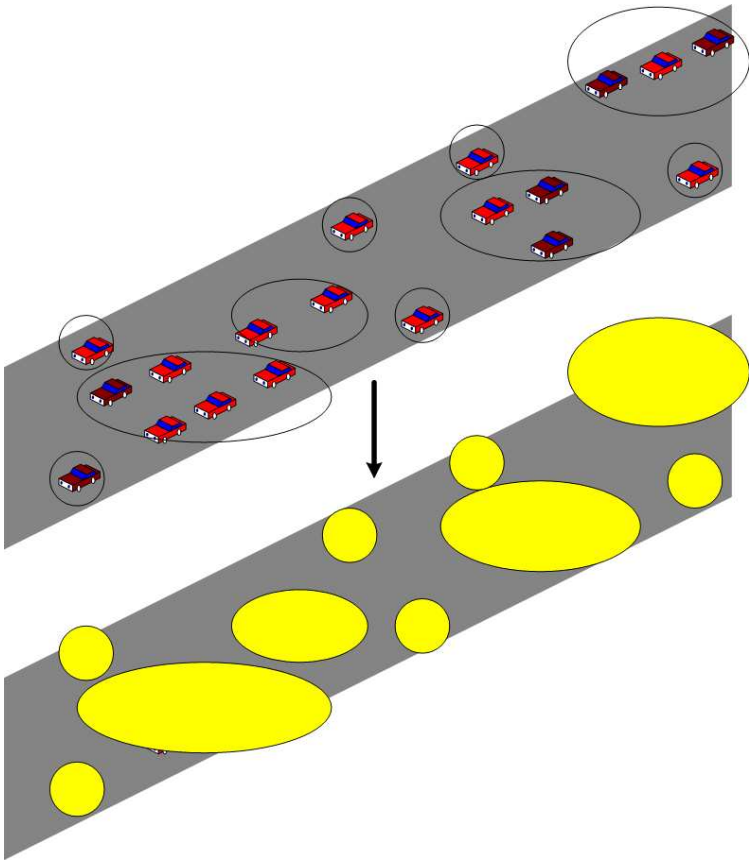


Figure 7.3: Clustering Cars on the Road Network

7.2 Moving Clusters

I employ a similar motion model as in [71, 54], where moving objects are assumed to move in a piecewise linear manner in a road network (Figure 7.4). Their movements are constrained by roads, which are connected by *network nodes*, also known as *connection nodes*.

Moving clusters can be represented by their centroid or by a set of distant points in a cluster [46]. Figure 7.2 depicts these two ideas. I use a centroid approach to represent the moving clusters in SCUBA. It works well when the clusters are compact[1].

I define centroid, radius and diameter for a cluster using the Euclidean distance as stated below. Given N d-dimensional data points in a cluster $\{\vec{X_i}\}$ where i = 1,...N, the **centroid** $\vec{X}0$, the **radius**

[1]For simplicity, I assume circular clusters, but the logic can be easily extended to other shapes (e.g., ellipses).

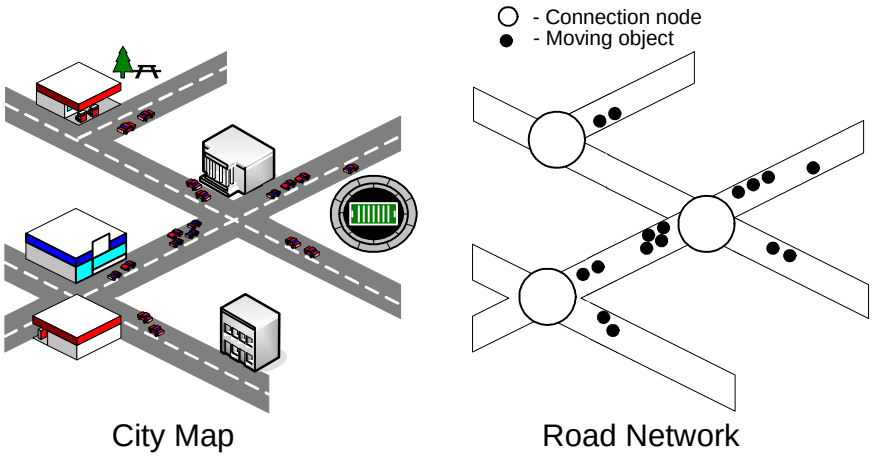


Figure 7.4: Road Network

R and **diameter** D of the cluster are defined as:

$$\vec{X0} = \frac{\sum_{\mathrm{i}=1}^{\mathrm{N}} \vec{\mathrm{X}}_{\mathrm{i}}}{N} \tag{7.1}$$

$$R = \left(\frac{\sum_{\mathrm{i}=1}^{\mathrm{N}} \left(\vec{\mathrm{X}}_{\mathrm{i}} - \vec{\mathrm{X}}0 \right)^2}{N} \right)^{\frac{1}{2}} \tag{7.2}$$

$$D = \left(\frac{\sum_{\mathrm{i}=1}^{\mathrm{N}} \sum_{j=1}^{N} \left(\vec{\mathrm{X}}_{\mathrm{i}} - \vec{\mathrm{X}}_{\mathrm{j}} \right)^2}{N\left(N-1\right)} \right)^{\frac{1}{2}} \tag{7.3}$$

R is the average distance from member points to the centroid. D is the average pairwise distance within a cluster. They are two alternative measures of the tightness of the cluster around the centroid.

I assume moving objects' location updates of the following form (*o.OID*, $o.Loc_t$, *o.t*, *o.Speed*, *o.CNLoc*, o.Attrs), where *o.OID* is the id of the moving object, $o.Loc_t$ is the position of the moving object, *o.t* is the time of the update, *o.Speed* is the current speed. I assume the speed doesn't change between two reported location updates. The *o.CNLoc* is the position of the connection node in the road network that will be passed by the moving object (its current destination). I assume that the

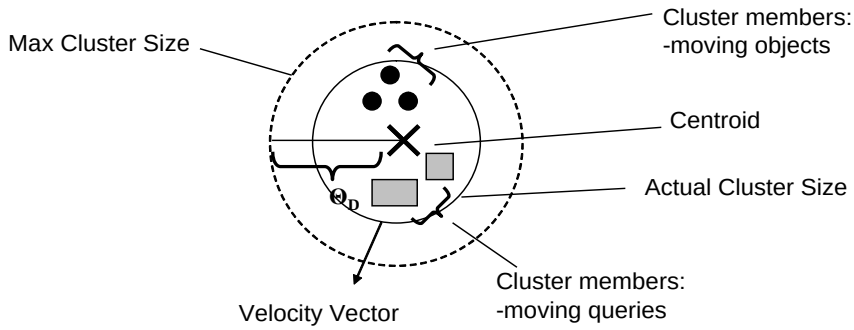


Figure 7.5: Moving Cluster in SCUBA

CNLoc of the object doesn't change before the object reaches this connection node. *o.Attrs* is a set of attributes describing the object (e.g., child, red car, etc.)

Similarly, the moving queries' location updates arrive via data stream, and have the following format (*q.QID*, $q.Loc_t$, *q.t*, *q.Speed*, *q.CNLoc*, *q.Attrs*), where *q.QID* is the id of the moving query, $q.Loc_t$ is the position of the query, *q.t* is the time of the update, *q.Speed* is the current speed, *q.CNLoc* is the position of the next node in the road network that will be passed by the moving query and the *q.Attrs* is a set of query-specific attributes (e.g., for range query it might be the size of the query, for *knn* query, the number of the nearest neighbors, etc.)

A moving cluster can contain both moving objects and moving queries (Figure 7.5). Moving objects and queries that don't satisfy conditions of any other existing clusters form their own clusters, *single-member moving clusters*. As objects and queries can enter or leave a moving cluster at any time, the properties of the cluster are adjusted accordingly. I consider the following attributes when grouping moving objects and queries into clusters: (1) Network constraint (e.g., road segment); (2) Speed; (3) Direction of the movement (e.g., connection node); (4) Relative spatial distance from each other.

A moving cluster *m* at time *t* is represented in the form (*m.CID*, $m.Loc_t$, *m.n*, *m.OIDs*, *m.QIDs*, *m.AveSpeed*, *m.CNLoc*, *m.R*, *m.ExpTime*), where *m.CID* is the moving cluster id, $m.Loc_t$ is the location of the centroid of the cluster at time *t*, *m.n* is the number of moving objects and queries that belong to this cluster, *m.OIDs* and *m.QIDs* are the collections of id's and relative positions of the moving objects and queries respectively that belong to this moving cluster, *m.AveSpeed* is

the average speed of the cluster, *m.CNLoc* is the cluster destination, *m.R* is the size of the radius, and *m.ExpTime* is the "expiration" time of the cluster (i.e., the time when the cluster reaches the *m.CNLoc* travelling at *m.AveSpeed*).

Being composed of objects and queries with similar properties, a moving cluster serves as a summary of its cluster members. Furthermore, we can treat a moving cluster just as another moving object that changes its location with time. The advantage of this is that all of the existing algorithms and indexing techniques proposed for moving objects can be easily extended to moving clusters.

The difficulty in maintaining and computing moving clusters is that once the clusters are formed at a certain time, with time clustering changes and deteriorates [38]. To keep a competitive and *high quality* clustering (i.e., clusters with compact sizes), I set the following thresholds to limit the sizes and deterioration of the clusters as the time progresses: (1) *distance threshold* (Θ_D), and (2) *speed threshold* (Θ_S). Distance threshold makes sure that the clustered entities are close to each other at the time of clustering. The speed threshold guarantees that the entities will stay close to each other for some time in the future.

Clusters are *dissolved* once they reach their destination points (road connection nodes). So if the distance between the location where the cluster has been formed is short, the clustering approach might be quite expensive and not as worthwhile. The same reasoning applies if the average speed of the cluster is very fast, and it reaches its destination point very quickly, forming a cluster might not give very little, if any, advantages. In a typical real-life scenario though, moving objects can reach relatively high speeds on the distant roads (e.g., highways), where connection nodes would be far apart from each other. On the smaller roads, the speed limits and the proximity of other cars would constrain the maximum speed the objects can develop, thus extending the time it takes for objects to reach the connection nodes. These observations support my intuition that clustering can be applicable to different speed scenarios for moving objects in every day life.

Individual positions[2] of moving objects and queries inside a cluster are represented in a relative

[2]In my implementation I store the coordinates of the cluster members in memory, but this can easily be changed to a secondary storage and accessed only during *join-within* operation.

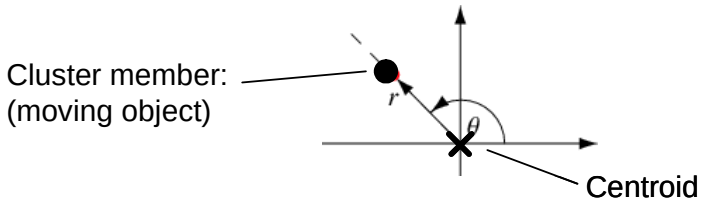


Figure 7.6: Cluster member representation inside cluster

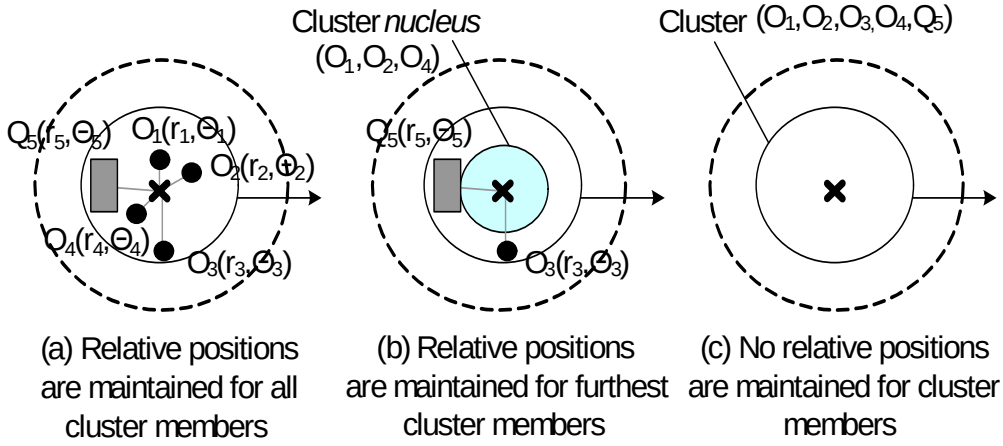


Figure 7.7: Handling cluster members

form using polar coordinates with the pole at the centroid of the cluster (Figure 7.6). For any location update point P, its polar coordinates are (r, θ), where r is the radial distance from the centroid, and θ is the the counterclockwise angle from the x-axis. As time progresses, the center of the cluster might shift, thus making it necessary to transform all the relative coordinates of the cluster members. I maintain a transformation vector for each cluster that records the changes in position of the centroid between the periodic executions. I refrain from constantly updating the relative positions of the cluster members, as it is not necessary, unless it is a *join-within* (Section 7.5.2). Before the *join-within* begins, the relative coordinates of the cluster members (that are being joined within a cluster) are translated from relative to the absolute positions.

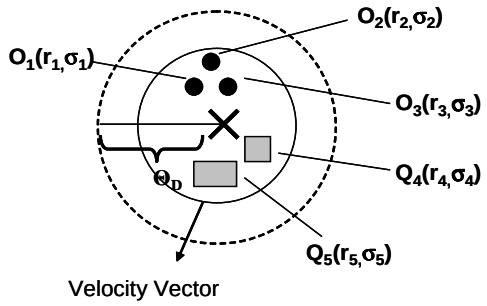


Figure 7.8: No load shedding

7.3 Moving Cluster-Driven Load Shedding

In this section I describe a technique to load-shed data based on moving clusters. As mentioned above, location-based services are characterized by a large number of objects and queries constantly sending their location updates. The arrival rates can be high and unpredictable. With limited resources, the system can potentially become overloaded. It may not be feasible to do a run-time distribution or add more resources. So the alternative would be to shed some data. A load shedding procedure should identify and discard the less important data (i.e., the data that would cause the minimal loss in accuracy of the answer).

Load shedding has been explored in networking [45], multimedia [18], and streaming databases [76, 75, 4]. In spatio-temporal databases, reduction of the amount of data is dealt with by controlling the update frequency [61, 81], where objects report their positions and velocity vectors only when their actual positions deviate from what they have previously reported by some threshold.

In this thesis, instead I explore a moving cluster-driven load-shedding technique. Specifically, I consider the data *inside* the moving clusters (i.e., the relative positions of the cluster members with respect to their centroids). As was described in Section 7.2, the individual positions of the cluster members are represented using polar coordinates relative to the centroid of the cluster.

Depending on the system load and the accuracy requirements, SCUBA can alternate between methods for handling internal members of the clusters (Figure 7.7). Namely, all cluster members' relative positions are maintained (Figure 7.8), none of the individual positions are maintained

(Figure 7.9), or a subset of relative positions of the cluster members (furthest from the centroid) are maintained (Figure 7.10). The rest of the members are abstracted into a structure inside a cluster called *nucleus*, a circular region that approximates the positions of the cluster members near the centroid of the cluster. The size of the nucleus is determined by the parameter Θ_N which is a fraction of the maximum size of the cluster determined by Θ_D. The size of the nucleus can be expressed as $\Theta_N = \frac{1}{\alpha} * \Theta_D$. The larger the value of α the smaller the nucleus, hence the more relative positions of the cluster members need to be maintained. Hence I load shed data based on common spatio-temporal attributes, maintained in the forms of moving clusters.

If the system is about to run out of memory, first, SCUBA foregoes maintenance for a subset of objects and queries' inside a cluster, and uses a *nucleus* to approximate the positions of the tightly-coupled around the centroid objects. If memory requirements are still high, then SCUBA can stop maintaining the relative positions of all cluster members altogether. In this case the cluster is the sole representative of the movement of the objects and queries that belong to it.

Such internal cluster representation trades off between accuracy and scalability. The accuracy highly depends on how compact the clusters are. The larger the size of the clusters, the more false positives we might get for answers when performing the join-between. If none of the relative positions are maintained, then when two clusters intersect (in *join-between*), we must assume that the objects from the clusters satisfy the queries from both clusters. Making the size of the clusters compact will give more accurate answers, but also will increase the overall number of clusters, and thus the join time. Increasing the size of clusters would make the processing faster, but with less

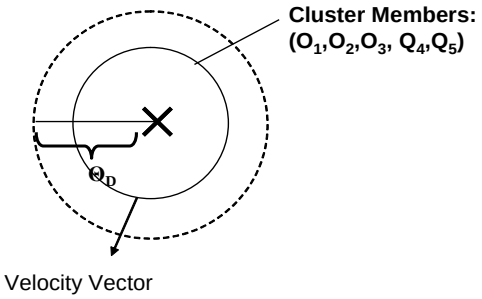


Figure 7.9: Full load shedding

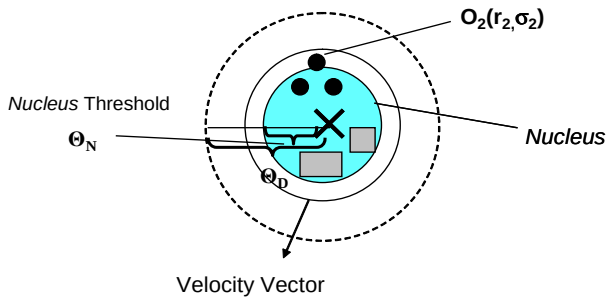


Figure 7.10: Partial load shedding

accurate results. In Section 8.1.5 I evaluate all three methods for handling the individual positions of the cluster members in terms of performance and accuracy.

7.4 Clustering Moving Objects

I use an incremental clustering algorithm based on the *Leader-Follower* clustering algorithm [39, 21] to create and maintain moving clusters in SCUBA. The major advantage of incremental clustering algorithms is that we don't have to store all the location updates (i.e., points) to form the clusters. So the space requirements are small. In addition, once Δ expires, SCUBA can immediately proceed with the join, without spending any time on re-clustering the entire data set. The disadvantage is that clustering is location update arrival order sensitive. I experimentally evaluate the tradeoff between the performance and clustering quality when clustering updates incrementally vs. non-incrementally, when the entire data set is available (Section 8.1.4).

When a location update from the moving object *o* arrives, I follow the following three steps to determine the moving cluster m it belongs to:

Step 1: Use moving object's position to probe the cluster grid table (Section 7.5.1) to find the moving clusters in the proximity of the current location of the object (i.e., clusters that the object can potentially join). If there are no clusters in the grid cell, then the object forms its own cluster, with the centroid at the current location update of the object, and radius = 0;

Step 2: If there are clusters that the object can potentially join, we iterate through the list of the clusters, and check the following properties:

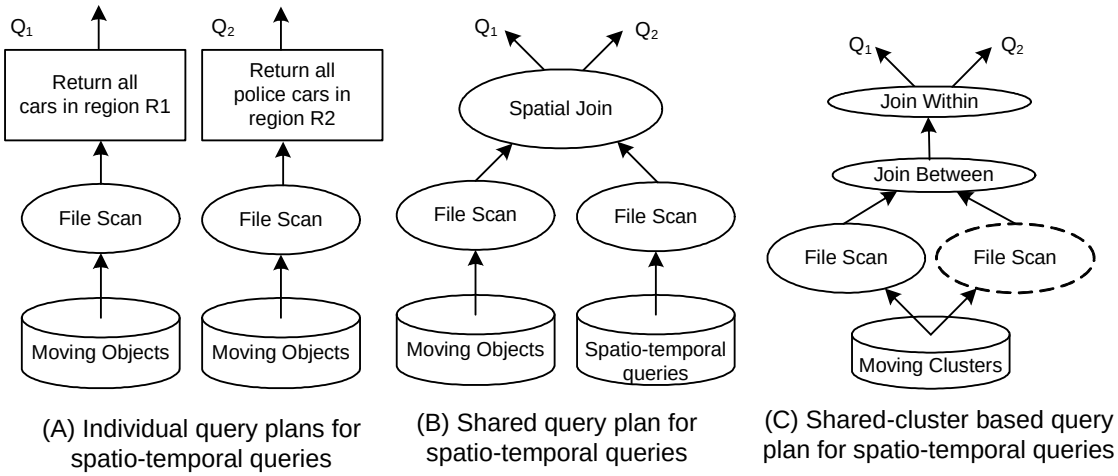


Figure 7.11: Shared execution

1. Is the moving object moving in the same direction as the cluster m ($o.CNLoc == m.CNLoc$)?

2. Is the distance between the centroid of the cluster and the location update less than the distance threshold $|o.Loc_t - m.Loc_t| \leq \Theta_D$?
3. Is the speed of the moving object less than the speed threshold $|o.Speed - m.AveSpeed| \leq \Theta_S$?

Step 3: If the moving object satisfies all three conditions in Step 2, then the moving cluster *absorbs* the moving object, and adjusts its (i.e., cluster) properties: the centroid, the average speed, the radius, and the count of the cluster members.

7.5 Shared Cluster-Based Processing

In this section, I present a *Scalable Cluster-Based Algorithm* (SCUBA) for evaluating continuous spatio-temporal queries on moving objects. SCUBA utilizes a *shared cluster-based execution paradigm* to reduce memory requirements and optimize the performance. The main idea behind shared cluster-based execution is to group similar objects as well as queries into moving clusters, and then the evaluation of a set of spatio-temporal queries is abstracted as a spatial join *between* the moving clusters and *within* the moving clusters.

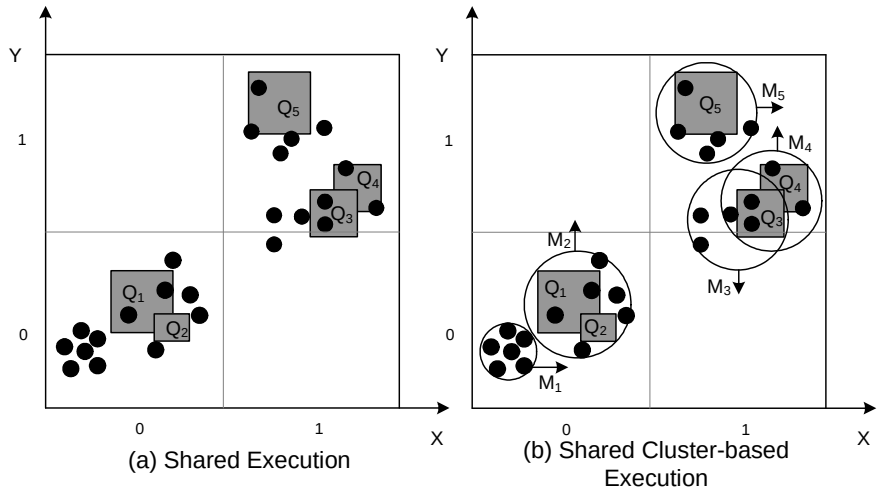


Figure 7.12: Join between moving objects and queries

To illustrate the idea, Figure 7.11 graphically depicts the difference between the traditional way to execute queries, the shared execution, and the shared-cluster based execution models. Traditionally a separate query plan is generated for each individual query (Figure 7.11(a)). Each query scans all the moving objects, filtering out only the ones that satisfy the predicate of the query. Figure 7.11(b) illustrates the *shared execution* paradigm where queries are grouped into a common query table, and then the problem of evaluating numerous spatio-temporal queries is abstracted as a spatial join between the set of moving objects and queries [84]. Having a shared plan allows only one scan over the moving objects. But we note that this method still joins queries and objects individually. With large numbers of objects and queries, this may still create a bottleneck in performance and potentially run out of memory. With *shared cluster-based execution* (Figure 7.11(c)), we form moving clusters, by grouping both moving objects and queries into heterogeneous moving clusters based on their common spatio-temporal attributes. Then a spatial join is performed on all moving clusters. Only if two clusters overlap, we have to go to the object/query level of processing, or given load shedding automatically assume that objects and queries within those clusters produce join results.

Figure 7.12 illustrates a motivating example for SCUBA. Here we assume that we maintain relative positions of all cluster members. With shared execution paradigm (Figure 7.12(a)) objects and queries would be joined as follows:

cell[0,0]:	2 queries $\bowtie$ 12 moving objects $\Rightarrow$ 24 ind. joins
cell[1,0]:	1 query $\bowtie$ 1 moving object $\Rightarrow$ 1 ind. join
cell[1,1]:	3 queries $\bowtie$ 11 moving objects $\Rightarrow$ 33 ind. joins
Total:	58 individual joins

With SCUBA (Figure 7.12(b)), we cluster objects and queries. We form 5 clusters: M_1(5 objects, 0 queries), M_2(6 objects, 2 queries), M_3(4 objects, 1 query), M_4(3 objects, 1 query), and M_5(5 objects, 1 query). So the execution would be as the following:

cell[0,0]:	2 clusters (M_1,M_2):
	2 *Between* joins and
	2 *Within* joins where:
	Join-Within(M_1) = 0 ind. joins
	Join-Within(M_2) = 12 ind. joins
cell[1,1]:	3 clusters (M_4,M_5,M_6)
	3 *Between* joins and
	4 *Within* joins where:
	Join-Within(M_3) = 3 ind. joins
	Join-Within(M_4) = 5 ind. joins
	Join-Within(M_5) = 5 ind. joins
	Join-Within(M_3,M_4) = 7 ind. joins
Total:	37 individual joins

The join between moving clusters M_3 and M_4 is done only once. Therefore these clusters are not joined in the cell[1,0]. So clearly, from the example above, fewer joins need to be made when utilizing SCUBA algorithm, thus minimizing the overall join time, and improving the performance.

7.5.1 Data Structures

In the course of execution, SCUBA maintains five in-memory data structures (Figure 7.14): (1) *ObjectsTable*, (2) *QueriesTable* (3) *ClusterHome*, (4) *ClusterStorage*, and (5) *ClusterGrid*.

ObjectsTable stores the information about objects and their attributes. An object entry in the *ObjectsTable* has the form (*o.OID*, *o.Attrs*), where *o.OID* is the object id, and *o.Attrs* is the list of attributes that describe the object. Similarly, the entry in the *QueriesTable* has the form (*q.QID*, *q.Attrs*) where *q.QID* is the query id, and the *q.Attrs* is the list of query attributes. *ClusterHome* is a hash table that keeps track of the current relationships between clusters and their members. A moving object/query can belong to only one cluster at a time *t*. An entry in the *ClusterHome* table is of the following form (*ID*, *type*, *CID*), where *ID* is the id of a moving entity, *type* indicates whether it's an object or a query, and *CID* is the id of the cluster that this moving entity belongs to. *ClusterStorage* is a table storing the information about all moving clusters currently present in the system (e.g., centroid, radius, member count, etc.). *ClusterGrid* is an in memory N x N grid table. The data space is divided into N x N grid cells. For each grid cell, *ClusterGrid* maintains a list of *CID*s (cluster ids) of moving clusters whose circular regions overlap with this cell.

7.5.2 The SCUBA Algorithm

In this section, I provide the overview and the details of the SCUBA algorithm.

The execution of SCUBA can be broken down into three phases: (1) Cluster Pre-Join Maintenance, (2) Cluster-Based Joining, and (3) Cluster Post-Join Maintenance phases (Figure 7.17).

In the cluster pre-join maintenance phase, the following are the tasks that take place: (1) formation of new clusters, (2) dissolving empty clusters, and (3) expansion of existing clusters.

In the cluster-based joining phase, we join clusters and then join objects and queries inside the overlapping clusters.

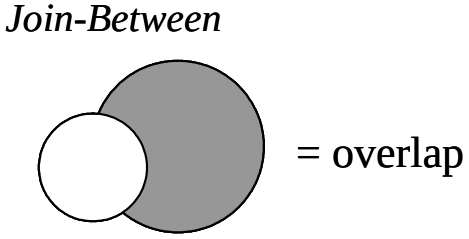


Figure 7.13: *Join-Between* clusters

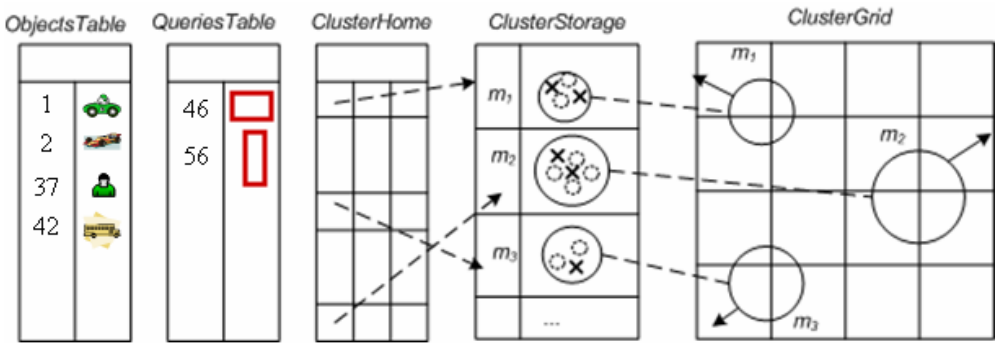


Figure 7.14: Data Structures used in SCUBA

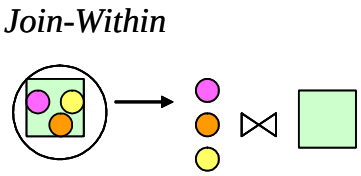


Figure 7.15: *Join-Within* for a singular cluster

In the cluster post-join maintenance phase, we dissolve expiring clusters and relocate non-expiring clusters based on a velocity-vector back into the grid.

Algorithm 6 gives the pseudo code for SCUBA.

For each execution interval, SCUBA first initializes the interval start time (Step 2). Before Δ time interval expires, SCUBA receives the incoming location updates from the moving objects and queries and incrementally updates existing moving clusters or creates new ones (Step 5). Alg. 7 gives the pseudo code for for the clustering procedure in SCUBA. I will use an example of a moving object when describing the clustering. Similar processing is done for queries. The algorithm starts

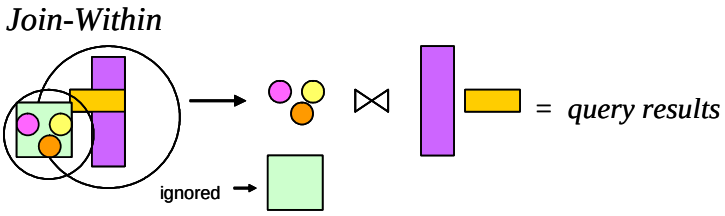


Figure 7.16: *Join-Within* for two clusters

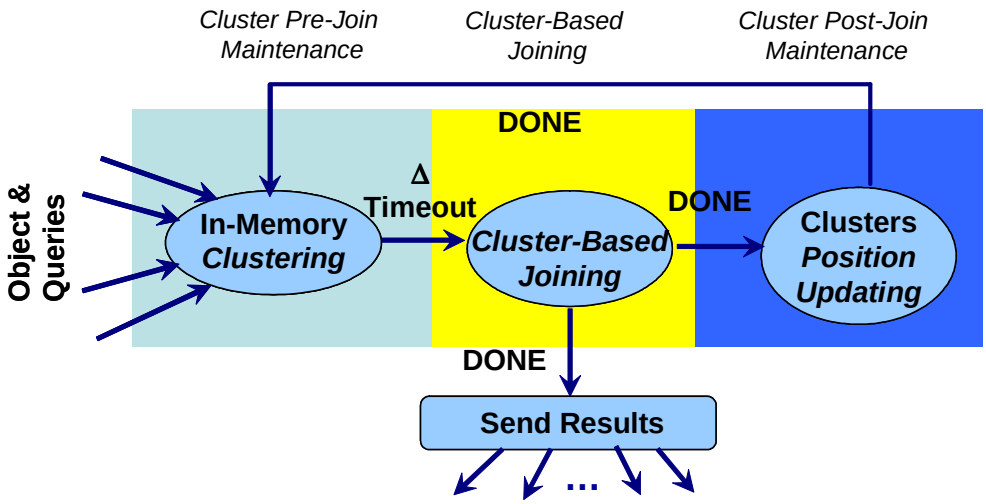


Figure 7.17: SCUBA state diagram

by checking the *ClusterHome* table to see if there is a cluster that the object belongs to already from the previous location updates (Algorithm 2, Step 2).

If there are none (this is a new object in the system), a new entry is made in the Objects table (Step 5). Location update is hashed to a grid cell, and a set of moving clusters is retrieved (currently overlapping with that grid cell). SCUBA checks each cluster to see if the current object can join that cluster (Steps 17-18). If yes, the object is *adopted* by the cluster (Step 19). Otherwise, the object forms its own cluster (Step 25).

If the object already belongs to a cluster due to previous location updates (Step 30), then SCUBA checks to see if the object is still related to this cluster (i.e., has similar attributes with the cluster) (Step 32). If it is still related, then it checks if the cluster expiration time (time when it reaches its destination) is greater than or equal to the time of the current location update (Step 35). The reason for this check is that we don't want to cluster the object to the cluster that expires before the timestamp of the current location update. If all tests are passed, SCUBA adjusts the object's relative position to the centroid based on the current location update (Step 37). If the dissimilarity between the object and the cluster has grown beyond thresholds (discussed in Section 7.4), other clusters are probed to see if the object can join them. If there are none, then the object forms its own cluster (Algorititthm 2, Steps 40-45).

Algorithm 6 *SCUBA()*

```
 1: loop
 2:    T_start = current time
 3:    while (current time - T_start) < Δ do
 4:       if new location update arrived then
 5:          Call MovingEntityClustering(source object o/query q)
 6:       end if
 7:    end while
       // Δ expires
       // begin query execution
 8:    for c = 0 to MAX_GRID_CELL do
 9:       for every moving cluster m_L ∈ G_c do
10:          for every moving cluster m_R ∈ G_c do
11:             //if the same cluster, do only join-within
12:             if (m_L == m_R) then
13:                Call DoWithinClusterJoin(m_L,m_R)
14:             else
15:                //do between-join only if 2 clusters contain members of different types
16:                if ((m_L.OIDs > 0) && (m_R.QIDs > 0)) ||
                   ((m_L.QIDs > 0) && (m_R.OIDs > 0)) then
17:                   if DoBetweenClusterJoin(m_L,m_R) == true then
18:                      Call DoWithinClusterJoin(m_L,m_R)
19:                   end if
20:                end if
21:             end if
22:          end for
23:       end for
24:    end for
25:    Send new query answers to users
26:    Call ClustersMaintenance() //do some cluster maintenance
27: end loop
```

When Δ time interval expires (location updating is done), SCUBA starts the query execution by performing *join-between* clusters (Algorithm 1, Step 11) and *join-within* clusters (Algorithm 1, Step 15). If two clusters are of the same type (all objects or all queries), they are not considered for the *join-between*. Similarly, if all of the members of a cluster are of the same type, no *join-within* is performed for that one cluster. The *join-between* checks if the circular regions of the two clusters overlap (Figure 7.13), and *join-within* performs a spatial join between the objects and queries inside each cluster, i.e., a self-cluster within join (Figure 7.15) and inside any two overlapping clusters (Figure 7.16). If *join-between* does not result in an intersection, then *join-within* can be skipped.

Algorithm 7 *MovingEntityClustering(moving entity (object or query) e)*

```
1:  m_H = A moving cluster from probing ClusterHome for e.ID
2:  if (m_H == null) then
3:    //entity doesn't belong to any cluster
4:    if e.type == object then
5:      Make new entry (e.ID, null) in ObjectsTable
6:    else if e.type == query then
7:      Make new entry (e.ID, null) in QueriesTable
8:    end if
9:    S_G = Set of moving clusters from probing ClusterGrid for e.Loc_t
10:   if (S_G == null) then
11:     //there are no other clusters in proximity of
        //current location update of entity e
12:     Call CreateNewCluster(e)
13:     return
14:   else
15:     //there are clusters that potentially this entity can join
16:     bool isClusterMember = false
17:     for every moving cluster m_G ∈ S_G do
18:       if (IsClusterCandidate(m_G, e) == true) then
19:         Call AdoptNewClusterMember(m_G, e)
20:         isClusterMember = true
21:         break
22:       end if
23:     end for
24:     if (isClusterMember == false) then
25:       Call CreateNewCluster(e)
26:       return
27:     end if
28:   end if
29: else
30:   //there is a cluster that moving entity already belongs to
      //(from previous update)
31:   bool isClusterMember = false
32:   if (IsStillClusterRelated(m_H, e) == false) then
33:     Call RemoveClusterMember(m_H, e.ID)
34:   else
35:     if (m_H.ExpTime ≥ e.t) then
36:       isClusterMember = true
          //adjust relative position of cluster member relative to centroid
37:       Call AdjustCurrentClusterMember(m_H, e)
38:     end if
39:   end if
40:   if isClusterMember == false then
41:     //check to see if there are other clusters the entity can join
        //(similar to steps 17-24 above)
        ...
        //still not a cluster member
42:     if isClusterMember == false then
43:       Call CreateNewCluster(e)
44:     end if
45:   end if
46: end if
47: return
```

7.6 Analysis of SCUBA

Here I analyze the performance of SCUBA in terms of memory requirements, number of join comparisons, and I/O cost. I use parameters listed in Table 7.1 in my analysis.

Memory Requirements: We maintain five in-memory data structures: $ObjectsTable$, $QueriesTable$,

Variable	Description
Δ	Time interval between periodic query execution
N_{obj}	Total number of objects
N_{qry}	Total number of queries
N_{clust}	Total number of clusters
A_{obj}	Average number of objects in a cluster $\approx \lceil N_{obj}/N_{clust} \rceil$
A_{qry}	Average number of queries in a cluster $\approx \lceil N_{qry}/N_{clust} \rceil$
A_{clust}	Average number of clusters in a grid cell $\approx \lceil N_{clust}/N_{cells} \rceil$
N_{UD}	Number of unique destinations on the road network
E_{obj}	Size of object entry in $Objects$ table
E_{qry}	Size of query entry in $Queries$ table
E_{CG}	Size of cluster entry in $ClusterGrid$
$E_{CS_{Base}}$	Size of cluster base entry (i.e., cid, centroid, radius) in $ClusterStorage$ table
$E_{CS_{CMember}}$	Size of cluster's member entry (i.e., id, relative position) in $ClusterStorage$ table
E_{CH}	Size of cluster entry in $ClusterHome$ table
$C_{overlap}$	Average number of grid cells that overlap with a cluster
S	Page size (measured in bytes)
N_{cells}	Number of grid cells
Θ_D	Distance threshold
Θ_S	Speed threshold
I_{ABJ}	Average percent of clusters intersecting with other clusters
α	Nucleus factor (from formula $\Theta_N = \frac{1}{\alpha} * \Theta_D$)
φ	Inverse load shedding factor
R_{AAI}	Average area of intersection when two clusters are intersected
R_{ACS}	Average cluster size (area)
C_x	Length of a grid cell in x dimension
C_y	Length of a grid cell in y dimension

Table 7.1: Parameters used in SCUBA analysis

$ClusterGrid$, $ClusterStorage$, and $ClusterHome$. The memory consumed by these structures can be described as follows. $ObjectsTable$ memory consumption is

$$ObjectsTable = N_{obj} * E_{obj} \tag{7.4}$$

Similarly, $QueriesTable$:

$$QueriesTable = N_{qry} * E_{qry} \tag{7.5}$$

$ClusterGrid$ memory requirements are

$$ClusterGrid = N_{clust} * C_{overlap} * E_{CG} \tag{7.6}$$

where $C_{overlap}$ describes the number of cells that a cluster overlaps with and can be estimated as

$$C_{overlap} = \left\lceil \frac{S_{clust}}{S_{cell}} \right\rceil \tag{7.7}$$

where S_{clust} is the largest possible cluster size and S_{cell} is a grid cell size. The size of a circular cluster is determined by the furthest away from the centroid cluster member. It can grow up to the distance threshold - Θ_D. S_{cell} represents the area of a rectangular cell, and can be described as S_{cell} = C_x* C_y, where C_x and C_y are the lengths of a grid cell in x and y dimensions. Hence we can rewrite (Equation 7.7) as

$$C_{overlap} = \left\lceil \frac{\pi * {\theta_D}^2}{C_x * C_y} \right\rceil \tag{7.8}$$

Equation (7.8) implies that $C_{overlap}$ depends on the size of the clusters and the size of the grid cells.

$ClusterStorage$ memory requirements are:

$$\begin{aligned} ClusterStorage &= N_{clust} * E_{CS_{Base}} + \\ &+ \varphi(N_{obj} * E_{CS_{CMember}} + N_{qry} * E_{CS_{CMember}}) \end{aligned} \tag{7.9}$$

where φ is an inverse load shedding factor ($0 \leq \varphi \leq 1$). When φ = 0, none of the relative location updates of cluster members are maintained. Thus, a moving cluster serves as a sole representation of the movements of objects and queries that belong to it. When φ = 1, all relative positions of the cluster members are preserved. In other words, no data is shed. When $0 < \varphi < 1$, partial load shedding is performed, that is (φ * 100)% of positions of the cluster members are maintained. So the lower the φ, the fewer positions are preserved (the greater the load shed occurs), and vice versa, the higher the φ, the more of the positions of the objects and queries inside a cluster are saved.

Memory consumed by $ClusterHome$ table is:

$$ClusterHome = N_{obj} * E_{CH} + N_{qry} * E_{CH} \tag{7.10}$$

Combining equations (7.4)-(7.10) and rearranging the terms, the total memory size consumed by

these structures is:

$$
\begin{aligned}
M_{SCUBA} &= N_{obj} * (E_{obj} + E_{CH}) + \\
&+ N_{qry} * (E_{qry} + E_{CH}) \\
&+ N_{clust} * (\left\lceil \frac{\pi * {\theta_D}^2}{C_x * C_y} \right\rceil * E_{CG} + E_{CS_{Base}}) + \\
&+ \varphi(N_{obj} + N_{qry}) * E_{CS_{CMember}}
\end{aligned}
\tag{7.11}
$$

Equation (7.11) suggests that once the environment parameters (E_{obj}, E_{qry}, E_{CH}, E_{CS}, and E_{CG}) are fixed, the total amount of memory consumed by SCUBA depends on the number of objects and queries, the number of clusters, the number of grid cells that the clusters overlap with and the load shedding factor. The number of clusters in SCUBA depends on the number of objects and queries, the two thresholds Θ_D and Θ_S, and the number of unique destinations that the clusters are moving towards (N_{UD}). $C_{overlap}$ depends on the size of the grid cells and the size of clusters. Assuming that size of the grid cells is fixed, then the size of the clusters has a direct effect on the memory consumption. Following the above observations, the clustering criteria, the distance (Θ_D) and speed thresholds (Θ_S) must be chosen carefully according to the application requirements and system configurations.

Faced with limited resources (e.g., memory, CPU) and near-real time response obligation when processing really bursty data streams, a call for load shedding can be justified. Yet load shedding comes at a price of losing accuracy. The accuracy (A_{SCUBA}) can be expressed as following:

$$
\begin{aligned}
A_{SCUBA} &= \max(\\
&(Nclust * \tfrac{R_{AAI}}{R_{ACS}} * (A_{obj} + A_{qry})), \\
&\varphi(N_{obj} + N_{qry}))/(N_{obj} + N_{qry})
\end{aligned}
\tag{7.12}
$$

I measure accuracy as the ratio of the number of relative positions of the cluster members maintained to the total number of objects and queries. The higher the number of positions preserved, the higher the accuracy of the results we expect. If φ = 1, the relative positions of all cluster members are preserved and accuracy = 100%. As φ decreases (i.e., load shedding increases) the accuracy decreases. If φ = 0 (i.e., all relative positions are load shed), the accuracy value is estimated by the ratio between the average intersection area and the total area of a cluster multiplied by the sum of

the average number of cluster members. Both equations (Equation 7.11) for memory and (Equation 7.12) for accuracy imply that as the inverse load shedding factor φ decreases, the memory requirements and accuracy decrease, and vice versa. This represents a tradeoff problem between the memory requirements and accuracy.

Join Performance: I analyze join performance in terms of the number of join comparisons. The number of joins is affected by the number of cluster members' positions maintained inside clusters (i.e., all, none, or some). If relative positions for all cluster members are maintained, the number of joins in SCUBA can be estimated as:

$$\begin{aligned} J_{SCUBA_{All}} &= N_{cells} * \left(\sum_{i=1}^{A_{clust}-1} i \right) + \\ &+(N_{clust} * A_{obj} * A_{qry})+ \\ &+(I_{ABJ} * N_{clust}) * A_{obj} * A_{qry} \end{aligned} \tag{7.13}$$

where $\sum_{i=1}^{A_{clust}-1} i$ is the average number of *between* cluster *joins* in a grid cell, N_{clust} *A_{obj} *A_{qry} is the number of individual joins done when performing a *within join* for a singular cluster (i.e., joining objects and queries that belong to the same cluster), and (I_{ABJ} *N_{clust}) * A_{obj} * A_{qry} is the number of individual joins performed for overlapping clusters (i.e., *join-within* for overlapping clusters). If the cluster member positions are not maintained the number of joins is

$$J_{SCUBA_{None}} = N_{cells} * \left(\sum_{i=1}^{A_{clust}-1} i \right) \tag{7.14}$$

This equals to the count of *join-between*'s only. Finally, if only some relative positions for cluster members are maintained and the rest of the cluster members are approximated by *nucleus*, the total number of joins is

$$\begin{aligned} J_{SCUBA_{Partial}} &= N_{cells} * \left(\sum_{i=1}^{A_{clust}-1} i \right) + \\ &+(N_{clust} * 1 * \tfrac{A_{obj}}{\alpha} * \tfrac{A_{qry}}{\alpha})+ \\ &+(I_{ABJ} * N_{clust}) * 1 * \tfrac{A_{obj}}{\alpha} * \tfrac{A_{qry}}{\alpha}) \end{aligned} \tag{7.15}$$

where "1" stands for one nucleus per cluster.

So the total number of join executions directly depends on the number of individual positions of

cluster members preserved, in other words the load shedding factor. If all of the relative positions of cluster members are shed, the number of joins depends on the average number of clusters per grid cell. This in its own turn depends on the total number of clusters, their size and the size of the grid cells. If some relative positions are maintained, the number of joins depends on the average number of clusters per cell, average number of objects and queries per cluster and the size of the nucleus (which is affected by the α), as well as the percent of clusters intersecting with each other. If all relative positions are kept, the number of joins depends on the average number of clusters per grid cell, average number of objects and queries per cluster and the percent of intersecting clusters.

I/O Cost: We execute SCUBA in-memory, but with an extremely large number of objects, queries and clusters and no load shedding allowed, another alternative would be spilling data to the disk. In particular, I focus on the $ClusterGrid$ table, as moving clusters frequently change their locations, and thus the grid would have to be frequently updated. The number of pages in each disk-based grid cell in the $ClusterGrid$ table is:

$$P_{GCell} = \left\lceil \frac{\left\lceil \frac{N_{clust}}{N_{cells}} \right\rceil}{\left\lfloor \frac{S}{E_{CG}} \right\rfloor} \right\rceil \tag{7.16}$$

So the total number of pages can be determined as:

$$P_{CG} = P_{GCell} * N_{cells} \tag{7.17}$$

The I/O cost[3] depends on the number of times SCUBA has to access the disk-based grid. This would happen in 3 cases:(1) when a new cluster is formed, (2) when a new cluster member increases the size of the cluster, and SCUBA has to check if the cluster might overlap with any surrounding grid cells, and (3) when doing a spatial join between moving clusters in a grid cell.

The cost of I/O when inserting a new cluster can be calculated as:

$$\begin{aligned} IO_{Insert} &= T_r + T_w * \\ &\min(P_{CG}, N_{clust} * C_{intersect} * P_{GCell}) \end{aligned} \tag{7.18}$$

where $C_{intersect}$ is the number of cells that intersect with a cluster inserted into the grid. T_r and T_w

[3]We measure the cost of I/O in terms of time.

denote the time to ”read” and ”write” to a grid cell respectively.

When the radius of the cluster increases, we need to update all the new grid cells that now the cluster might overlap with. So the I/O cost for expansion of the cluster can be approximated as:

$$\begin{aligned} &IO_{ClustExp} = T_r + T_w* \\ &\min(P_{CG}, C_{NewOverlap} * P_{GCell})+ \\ &+T_r * T_w * \rho * \Delta * (r_{obj} + r_{qry}) \end{aligned} \tag{7.19}$$

where $C_{NewOverlap}$ is the number of new cells that the cluster overlaps with, as a result of cluster expansion. Since the expansion of the clusters is affected by the arrival of location updates from objects and queries, this introduces the part T_r*T_w*ρ*Δ*(r_{obj} + r_{qry}) in the Equation 7.19. ρ is the percent of objects and queries causing the expansion of the cluster. If the update falls within the current cluster without causing the increase in size, no access to the grid is necessary. If the update causes the expansion of the cluster, we have to look up the grid for the current position of the centroid of the cluster, and record the expansion of the cluster in the grid.

When doing the join between moving clusters, each grid cell would be read only once. So the I/O cost for the join is:

$$IO_{join} = T_r * \min(P_{CG}, C_{overlap} * N_{clust} * P_{GCell}) \tag{7.20}$$

So the overall cost of the I/O in SCUBA is the sum of of the I/O costs from the insertion of clusters into the grid, expansion of clusters in the grid, and joining moving clusters in the grid.

$$\begin{aligned} &IO_{SCUBA} = T_r * T_w* \\ &\min(P_{CG}, N_{clust} * C_{intersect} * P_{GCell})+ \\ &+T_r * T_w * \min(P_{CG}, (C_{NewOverlap} * P_{GCell}))+ \\ &+T_r * T_w * \rho * \Delta * (r_{obj} + r_{qry})+ \\ &+T_r * \min(P_{CG}, (C_{overlap} * N_{clust} * P_{GCell})) \end{aligned} \tag{7.21}$$

I assume that $T_w > T_r$, then T_r can be ignored. So the upper bound of the total I/O cost is

$$\begin{aligned} &IO_{SCUBA} = 4 * T_w * P_{CG}+ \\ &+2 * T_w * \rho * \Delta * (r_{obj} + r_{qry}) \end{aligned} \tag{7.22}$$

This means that the total number of I/Os for SCUBA depends on: (1) the time to write to a grid cell, (2) the total number of pages in the $ClusterGrid$, (3) the length of the time interval Δ, (4) the arrival rate of objects and queries, and (5) the percent of the moving objects and queries that cause the expansion of the clusters they belong to. We note also that ρ depends on the thresholds we specify for clustering (Θ_D and Θ_S) and size of the grid cells.

Chapter 8

Experimental Results: SCUBA Evaluation

In this chapter I describe experimental results evaluating the performance of SCUBA compared to regular (non-clustered) grid-based evaluation of spatio-temporal queries.

8.1 Experimental Evaluations

In this section, I describe my experimental evaluations of SCUBA. I compare SCUBA with a grid-based spatio-temporal range algorithm[1], where objects and queries are hashed based on their locations into different grid cells. Then a cell-by-cell join between moving objects and queries is performed. Grid-based execution approach is a common choice for spatio-temporal query execution. Many works in the literature exploit it in one way or the other (e.g., [57, 84, 62, 22, 60]). In all of the experiments queries are evaluated periodically (every Δ time units).

To be fair in comparison, I implemented incremental hashing for the regular execution model, where objects and queries are immediately inserted into the grid upon their arrival.

Section 8.1.1 describes my experimental settings. Section 8.1.2 compares the performance of SCUBA with the regular execution model when varying grid cell sizes. In Section 8.1.3, I study the performance of SCUBA with varying cluster skewing factors. Section 8.1.4 studies the performance of SCUBA when performing incremental vs. non-incremental clustering. Finally,

[1]For simplicity, I will refer to it as *regular execution* or *regular operator*.

Section 8.1.5 studies the performance and accuracy of SCUBA when performing load shedding on arriving location updates.

8.1.1 Experimental Settings

I have implemented SCUBA inside our stream processing system CAPE [64]. Moving objects and moving queries generated by the *Network-Based Generator of Moving Objects* [10] are used as data. The input to the generator is the road map of Worcester, USA. All the experiments were performed on Red Hat Linux (3.2.3-24) with Intel(R) XEON(TM) CPU 2.40GHz and 2GB RAM. Unless mentioned otherwise, the following parameters are used in the experiments. The set of objects consists of 10,000 objects and 10,000 spatio-temporal range queries. Each evaluation interval, 100% of objects and queries send their location updates. Relative positions of all cluster members are maintained inside clusters (i.e., no load shedding). For the *ClusterGrid* table I chose 100x100 grid size. Δ is set to 2 time units. The distance threshold Θ_D equals 100 spatial units. The speed threshold Θ_S is set to 10 (spatial units/time units).

8.1.2 Varying Grid Cell Size

In this section, I compare the performance and memory consumption of SCUBA and the regular grid-based algorithm when varying the grid cell size. Figure 8.1 gives the effect of increasing the granularity of the grid in regular and SCUBA operator. Since the coverage area (the city of Worcester) is constant, by increasing/decreasing the cell count in each dimension (x- and y-), we can control the sizes of the grid cells. So in 50x50 grid the size of a grid cell is larger than in 150x150 grid. So the larger the count of the grid cells, the smaller they are in size and vice versa.

From Figure 8.1a, the join time decreases for the regular operator significantly when decreasing the grid cell size. The reason for that is that smaller cells contain fewer objects and queries. Hence, fewer comparisons (joins) need to be made. But the fine granularity of the grid comes at a price of higher memory consumption. This is due to the fact that a large number of grid cells are created, each containing individual location updates of objects and queries.

The join time for SCUBA slightly goes up as the grid cell sizes become smaller. But the change

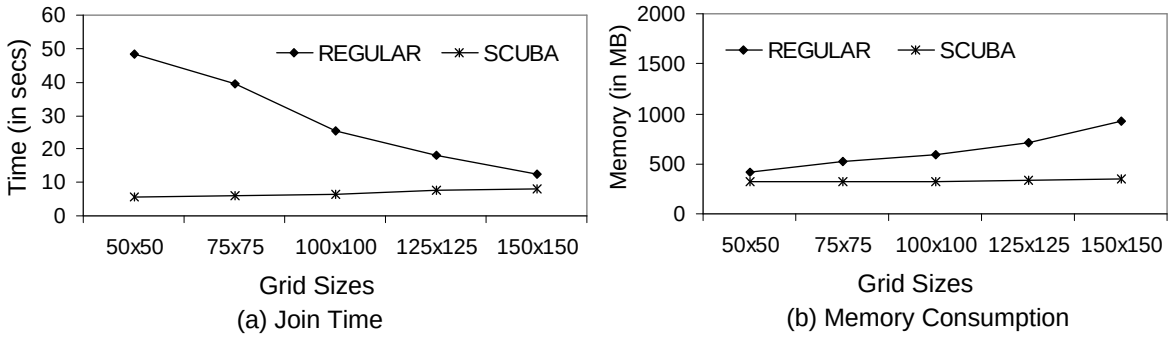


Figure 8.1: Varying grid size

is insignificant, because the cluster sizes are compact and even as the granularity of the cells is increasing, the size of grid cells is still larger than the size of clusters. So unless many clusters are on the borderline of the grid cells (overlapping with more than one cell), the performance of SCUBA is not ”hurt” by the finer granularity of the grid. Moreover, only one entry per cluster (which aggregates several objects and queries) needs to be made in a grid cell vs. having an individual entry for each object and query. This provides major memory savings when processing extremely large numbers of densely moving objects and queries.

8.1.3 Varying Skewness

In this section, I compare the performance of SCUBA with regular grid-based method when skewing the spatio-temporal attributes of moving objects and queries. I vary the attributes causing objects and queries to be very dissimilar (no common attributes) or very much alike (i.e., clusterable). Several data sets were generated varying clusterable attributes of moving objects and queries (e.g., speed, destination, road network) to vary the number of clusters and the number of cluster members per cluster.

Figure 8.3 illustrates the effect of skewing the data from being very dissimilar to very alike in movement. When data is dissimilar, many single member clusters or clusters with few cluster members are formed. When data has many similarities, few clusters are formed containing a large number of cluster members. The *skew factor* represents the average number of moving entities that have similar spatio-temporal properties, and thus could be grouped into one cluster. For instance, when skew factor = 1, each object and query moves in a distinct way. Hence each forms its own

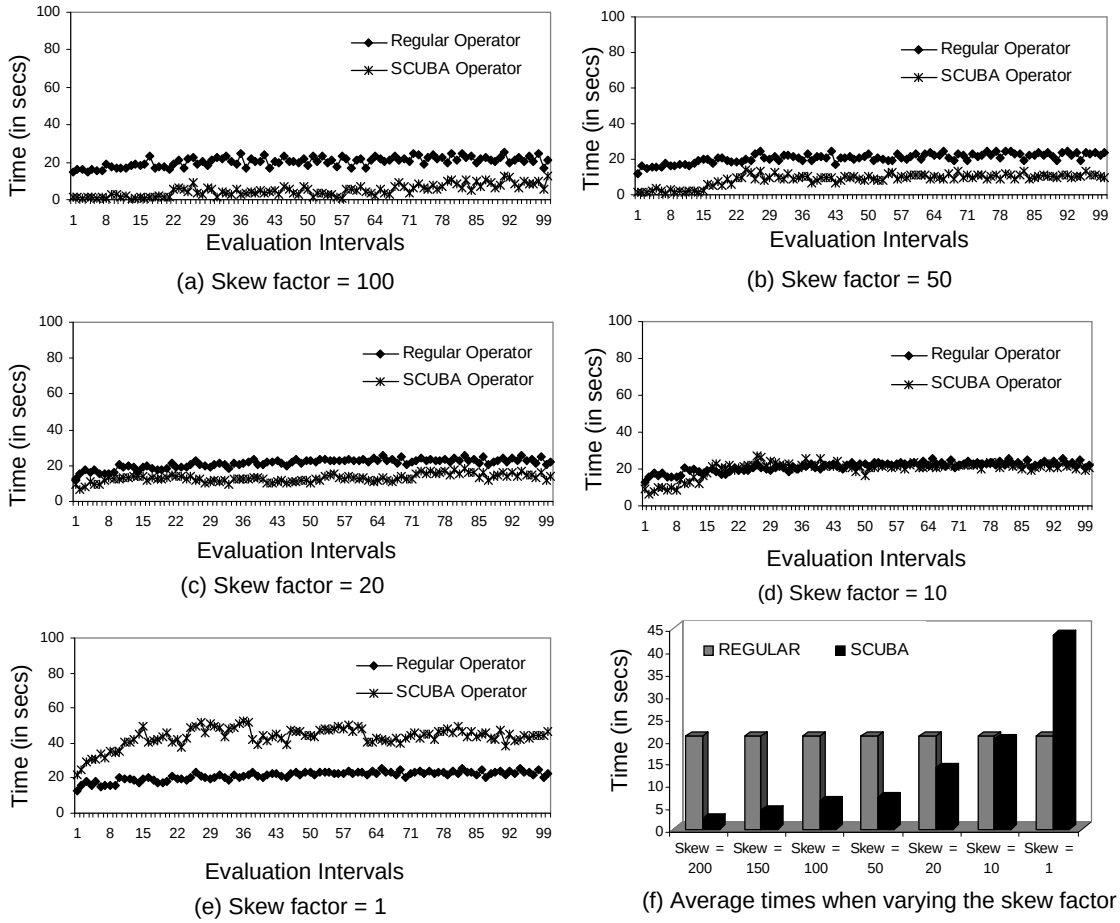


Figure 8.2: Shapshots of execution intervals with varying skewing factor

cluster. When the skew factor = 100, every 100 objects/queries sending their updates move in a similar way. Thus they typically may form a cluster. I kept the relationship between objects/queries and clusters they belong to unchangeable until the cluster gets dissolved (i.e., once an object/query belongs to a cluster, it doesn't leave it).

In Figure 8.3, when objects and queries are non-clusterable, the SCUBA performance suffers due to the overhead of many single-member clusters where many join-between clusters are performed. If many single member clusters spatially overlap, the join-within is performed as well. This increases the overall join time. In real life this scenario is highly unlikely as with large number of moving objects there often may be at least some that would have common motion attributes for some duration of time. As the skew factor increases (10-100), and more objects and queries are clusterable, the join time for SCUBA significantly decreases. The overall join time is roughly

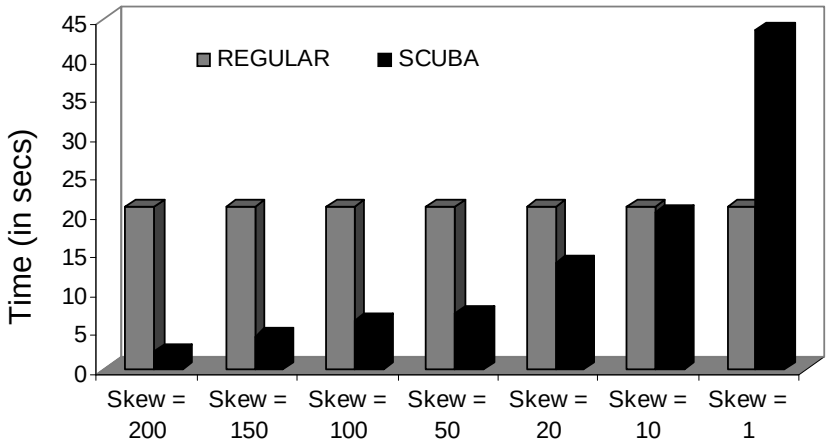


Figure 8.3: Join time with skewing factor

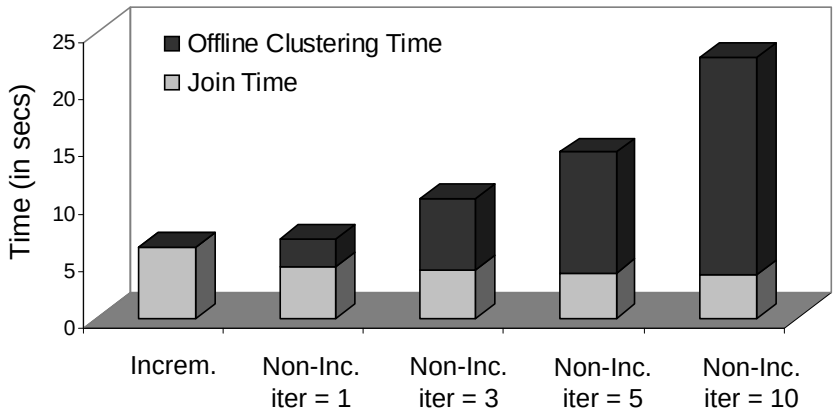


Figure 8.4: Incremental vs. Non-Incremental Clustering

3.5 times faster compared to a regular grid-based approach when the skew factor equals 100, i.e., approximately 100 moving entities per cluster.

8.1.4 Incremental vs. Non-incremental Clustering

In this section I study the tradeoff between the improved quality of the clusters which can be achieved when clustering is done non-incrementally (when all data points are available) and the performance of SCUBA. As proposed, SCUBA clusters location updates incrementally upon their arrival. The quality of the clusters is affected by the order in which the updates arrive. I wanted to investigate if clustering done offline (i.e., non-incrementally, when all the data points are available)

producing better quality cluster groups and facilitating a faster join-between the clusters could outweigh the cost of non-incremental clustering. In particular, I focus on the join processing time, and how much improvement in join processing could be achieved with better quality clusters vs. when clusters are of slightly poorer quality when formed incrementally.

I implemented a *K-means* extension to SCUBA for non-incremental clustering. One of the disadvantages of the K-means algorithm is that the number of clusters must be decided in advance. I used a tracking counter for the number of unique destinations of objects and queries for a rough estimate of the number of clusters needed. Another disadvantage is that K-means needs several iterations over the dataset before it converges. With each iteration, the quality of clustering improves, but the clustering time takes significantly longer. I varied the number of iterations from 1 to 10 in this experiment to observe the impact on quality of clusters by increased number of iterations.

Figure 8.4 presents the join times for SCUBA when clustering is done incrementally vs non-incrementally. The bars represent a combined cost of clustering time and join time. The time to perform incremental clustering is omitted as the join processing starts immediately when Δ expires. In the offline clustering scenario, the clustering has to be done first before proceeding to the join. With the increased number of iterations, the quality of clusters is better. This aids in faster join execution compared to the incremental case. The cost of waiting for the offline algorithm to finish the clustering outweighs the advantage of the faster join. When the number of iterations is 3 or greater, the clustering time takes longer than the actual join processing. The larger the dataset the more expensive each iteration becomes. Offline clustering is not suitable for clustering large amounts of moving objects when there are constraints on execution time and memory space. Even with a reduced number of scans through the data set and improved join time, the advantage of having better quality clusters is not amortized due to the amount of time spent on offline clustering and larger memory requirements.

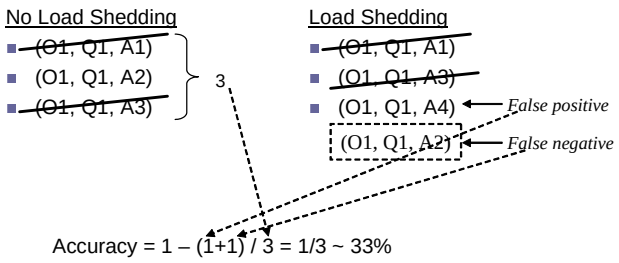


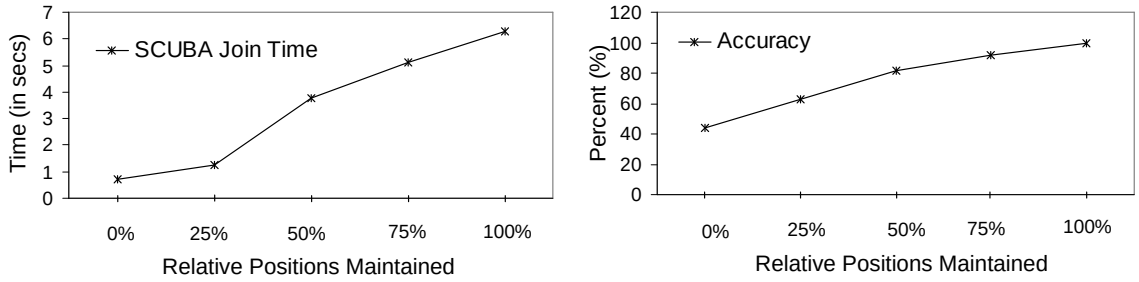
Figure 8.5: Measuring accuracy when performing load shedding

8.1.5 SCUBA and Load Shedding

Here I evaluate the effect of moving cluster-based load shedding on performance and accuracy in SCUBA.

Figure 8.6 represents the join processing time and accuracy measurements for SCUBA when load shedding positions of the cluster members. The x-axis represents the percent of the size of the *nucleus* (i.e., circular region in the cluster containing cluster members whose positions are discarded) with respect to the maximum size of the cluster. For simplicity, I will refer to it as η. When η = 0%, no data is discarded. On the opposite, when η = 100%, all cluster members' positions are discarded, and the cluster solely approximates the positions of its members. The fewer relative positions are maintained, the fewer individual joins need to be performed when executing a join-within for overlapping clusters.

Our experiments illustrate that this load shedding comes at a price of less accurate results (Figure 8.6b). To measure accuracy, I compared the results outputted by SCUBA when η = 0% (no load shedding) to the ones output when $\eta > 0$%, calculating the number of false-negative and



(a) Join time when load shedding

(b) Accuracy when load shedding

Figure 8.6: Cluster-Based Load Shedding

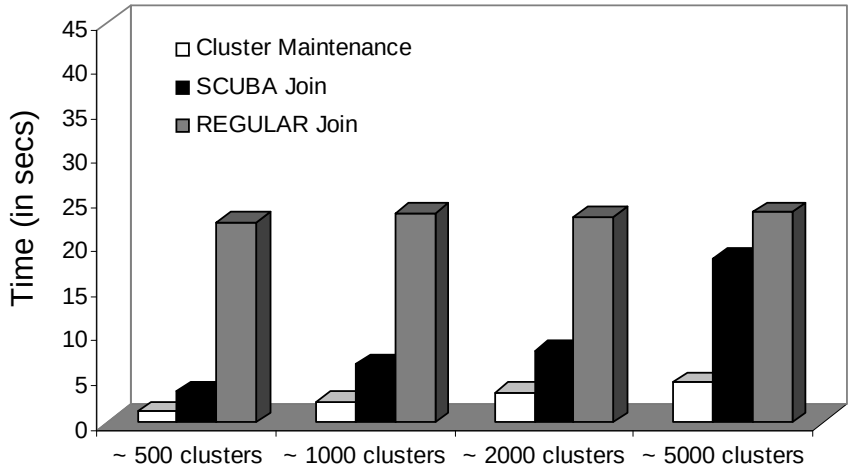


Figure 8.7: Cluster Maintenance

false-positive results (Figure 8.5). The size of the nucleus has a significant impact on the accuracy of the results when performing load shedding. Hence it must be carefully considered. When η = 50%, the accuracy $\approx$ 79%. So relatively good results can be produced with cluster-based load shedding.

Spatio-temporal applications are often more sensitive to changes in data characteristics than regular streaming applications. For instance, a random drop of location updates when objects move especially fast may totally change the results.

8.1.6 Cluster Maintenance Cost

In this section, I compare the cluster maintenance cost with respect to join time in SCUBA and regular grid-based join time. Figure 8.7 gives the cluster maintenance time when the number of clusters is varied. By cluster maintenance cost we mean the time it takes to pre- and post-process the clusters before and after the join is complete (i.e., form new clusters, expand existing clusters, calculate the future position of the cluster using its average speed, dissolve expired clusters, and re-insert clusters into the grid for the next evaluation interval).

For this experiment, I varied the skew factor to affect the average number of the clusters. The x-axis represents the average number of clusters in the system. Figure 8.7 shows that the

cluster maintenance (which is an overhead for SCUBA) is relatively cheap. If we combine cluster maintenance with SCUBA join time to represent the overall join cost for SCUBA, it is still faster than the regular grid-based execution. Hence, even though maintaining clusters comes with a cost, it is still cheaper than keeping the complete information about individual locations of objects and queries and processing them individually.

Chapter 9

Part IV:

Conclusions and Future Work

9.1 Conclusions

This thesis focuses on the understanding of the tradeoff in performance and accuracy when deciding on a location modelling technique for spatio-temporal queries on moving objects. Two types, namely discrete and continuous location management models have been studied. An accuracy comparison technique for the discrete and continuous results was developed. This allowed to compare otherwise very different types of results for the two models. Experiments and accuracy measurements of the output results have shown that when objects move slowly, the discrete model doesn't "miss" on many results (i.e., intersections between objects and queries), hence the results stay relatively accurate compared to the continuous model. But as the speed increases, the discrete model begins to "miss out" on many results that the continuous model doesn't. Hence the accuracy of the discrete location model decreases as the speed of the moving objects increases. On the other hand, the continuous model can produce more accurate results, but at a high cost of join execution. This is due to large distances travelled between the location updates, which have to be "clipped" (i.e., intersected with) the grid cells of the index, causing many overlaps, and thus increasing join

time. So in the fast moving environments, higher accuracy can be achieved with the continuous model, but at the cost of higher join time. Another conclusion from the experiments is that if a server cannot catch up with the data and a load shedding is performed, the continuous model can still produce relatively high accuracy results. In other words, if the application requirements are the highest possible accuracy, then the continuous model should be used. If the application needs the best possible performance at the cost of accuracy, then discrete model should be used. The grey area is when both high performance and best possible accuracy are requested. In this case, it's a choice between the discrete model with 100% of location updates and the continuous model with some load shedding (e.g., load shedding rate α = 20%, which means 20% of data will be dropped, and only 80% processed). This value of 80% corresponds to update probability parameter (in our experiments). Calculating the accuracy value and modifying the load shedding rate at run-time would allow to tune the system to the best possible accuracy/performance ratio. This dynamic behavior is one of the future tasks for implementation.

Also in this thesis, I proposed, SCUBA, a *Scalable Cluster Based Algorithm* for evaluating a large set of spatio-temporal queries continuously. SCUBA combines clustering analysis with shared execution paradigm and moving cluster-based load shedding to achieve improvement in performance when evaluating moving queries on moving objects. Given a set of moving objects and queries, SCUBA groups them into moving clusters based on common spatio-temporal attributes. The clusters serve as abstractions of data, and can be used to optimize the join processing. SCUBA performs a spatial join between the moving clusters and only if the two join (i.e., intersect), the join-within clusters is executed. Hence only if two clusters intersect, the execution has to go to the individual level of processing (i.e., joining objects with queries).

Clusters can serve as approximations for the locations of its members. As a consequence, a more "intelligent" load shedding can be performed using moving clusters in case the system has to drop some of its workload. The contribution of this thesis is the utilization of moving clustering techniques as means to optimize internal execution of continuous queries in data streaming systems. Experimental results show that SCUBA is efficient at processing large numbers of concurrent

range queries on a large number of moving objects. It shows to be superior than the alternative of individual processing of each query when large numbers of moving objects move in similar fashion (i.e., on the same network, at relatively similar speeds, and in the same direction).

9.2 Future Work

For future work, we plan to extend our study in the following directions.

First, I assume only circular clusters. A natural extension would be to allow non-circular clusters.

Currently, the parameters for the maximum sizes of the moving clusters and the sizes of the nuclei are fixed. The next step would be to dynamically adjust the sizes of the clusters and nuclei (which indirectly determine the amount of data to be load shed) dynamically at run-time responding to the system status (i.e., statistics). For example, if the system is not catching up with the data and a larger amount of data needs to be load shed, SCUBA can dynamically adjust the nuclei sized for all (or ideally for some - least affected) moving clusters to speed up the processing while still maintaining the highest possible accuracy.

Another possible extension, would be to utilize other indexing techniques with SCUBA. In this thesis, a grid index was utilized with SCUBA, but possibly other index variations can result in even more efficient processing when combined with SCUBA.

Currently the centroids of the moving clusters are computed for high-density regions (i.e., many moving objects are attempted to be clustered together in the same cluster, and a centroid is recomputed every time a new point is added). This may become very expensive as the number of data points joining the cluster increases. It might be better to cluster by using low-density regions (i.e., regions that don't contain any objects), to define the boundaries between the clusters, rather than using high-density areas to define the centers of the clusters.

We will also plan to extend SCUBA algorithm for the processing of other types of spatio-temporal queries (e.g., knn, aggregate, etc.). Since moving clusters group objects by similar prop-

erties, they may naturally facilitate a faster execution for different types of queries. For example, finding a nearest moving neighbor for each moving object, or finding similarly moving groups of objects for data analysis, etc.

Another area of further optimization of the algorithm would be to allow merging and breaking clusters and supporting hierarchical clustering to further optimize the processing. In addition, we plan to test SCUBA on moving objects and queries using real physical location-monitoring sensors.

Also as part of our future work, we plan to experimentally evaluate dynamically adaptive spatio-temporal operator behavior, utilizing the accuracy/performance tradeoff model. The spatio-temporal operator would dynamically switch between the two models depending on the current data arrival rates and the average velocities of the objects and queries, to maximize the accuracy and minimize the cost of execution.

Bibliography

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. F. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. B. Zdonik. Aurora: A data stream management system. In *SIGMOD Conference*, page 666, 2003.

[2] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *PODS*, pages 175–186, 2000.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[4] B. Babcock, M. Datar, and R. Motwani. Load shedding techniques for data stream systems, 2003.

[5] D. Barbará. Chaotic mining: Knowledge discovery using the fractal dimension. In *1999 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, 1999.

[6] D. Barbará. Requirements for clustering data streams. *SIGKDD Explorations*, 3(2):23–27, 2002.

[7] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *VLDB J.*, 5(4):264–275, 1996.

[8] P. Berkhin. Survey of clustering data mining techniques. Technical report, Accrue Software, San Jose, CA, 2002.

[9] C. M. Bishop. *Neural networks for pattern recognition*. Oxford University Press, Oxford, UK, UK, 1996.

[10] T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.

[11] Y. Cai, K. A. Hua, and G. Cao. Processing range-monitoring queries on heterogeneous mobile objects. In *Mobile Data Management*, pages 27–38, 2004.

[12] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing. In *SIGMOD Conference*, page 668, 2003.

[13] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *VLDB*, pages 203–214, 2002.

[14] S. Chandrasekaran and M. J. Franklin. Psoup: a system for streaming queries over streaming data. *VLDB J.*, 12(2):140–156, 2003.

[15] B. B. Chaudhuri. Dynamic clustering for time incremental data. *Pattern Recognition Letters*, 15(1):27–34, 1994.

[16] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD*, pages 379–390, 2000.

[17] A. Civilis, C. S. Jensen, J. Nenortaite, and S. Pakalnis. Efficient tracking of moving objects with precision guarantees. In *MobiQuitous*, pages 164–173, 2004.

[18] C. L. Compton and D. L. Tennenhouse. Collaborative load shedding for media-based applications. In *International Conference on Multimedia Computing and Systems*, pages 496–501, 1994.

[19] C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *SIGMOD Conference*, pages 647–651, 2003.

[20] P. Domingos and G. Hulten. Catching up with the data: Research issues in mining data streams. In *DMKD*, 2001.

[21] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley-Interscience Publication, 2000.

[22] H. G. Elmongui, M. F. Mokbel, and W. G. Aref. Spatio-temporal histograms. 2005.

[23] D. H. Fisher. Iterative optimization and simplification of hierarchical clusterings. *CoRR*, cs.AI/9604103, 1996.

[24] L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. A data model and data structures for moving objects databases. In *SIGMOD Conference*, pages 319–330, 2000.

[25] M. M. Gaber, A. B. Zaslavsky, and S. Krishnaswamy. Towards an adaptive approach for mining data streams in resource constrained environments. In *DaWaK*, pages 189–198, 2004.

[26] B. Gedik and L. Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, pages 67–87, 2004.

[27] F. Geerts. Moving objects and their equations of motion. In *CDB*, pages 41–52, 2004.

[28] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *FOCS*, pages 359–366, 2000.

[29] L. Guibas. Kinetic data structures: A state of the art report, 1998.

[30] C. Gupta and R. L. Grossman. Genic: A single-pass generalized incremental algorithm for clustering. In *SDM*, 2004.

[31] S. K. Gupta, K. S. Rao, and V. Bhatnagar. K-means clustering algorithm for categorical attributes. In *DaWaK*, pages 203–208, 1999.

[32] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representing and quering moving objects. *ACM Trans. Database Syst.*, 25(1):1–42, 2000.

[33] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[34] M. Hadjieleftheriou, G. Kollios, D. Gunopulos, and V. J. Tsotras. On-line discovery of dense areas in spatio-temporal databases. In *SSTD*, pages 306–324, 2003.

[35] S. E. Hambrusch, C.-M. Liu, W. G. Aref, and S. Prabhakar. Query processing in broadcasted spatial index trees. In *SSTD*, pages 502–521, 2001.

[36] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB*, pages 297–308, 2003.

[37] M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. S. Marzouk, and X. Xiong. Nile: A query processing engine for data streams. In *ICDE*, page 851, 2004.

[38] S. Har-Peled. Clustering motion. In *FOCS '01: Proceedings of the 42nd IEEE symposium on Foundations of Computer Science*, page 84, Washington, DC, USA, 2001. IEEE Computer Society.

[39] J. A. Hartigan. *Clustering Algorithms*. John Wiley and Sons, 1975.

[40] T. Hastie, R. Tibshirani, and J. H. Friedman. *The elements of statistical learning: data mining, inference, and prediction: with 200 full-color illustrations*. New York: Springer-Verlag, 2001.

[41] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1994.

[42] K. Hornsby and M. J. Egenhofer. Modeling moving objects over multiple granularities. *Ann. Math. Artif. Intell.*, 36(1-2):177–194, 2002.

[43] A. Huber. *Incremental Pattern Recognition in Data Streams*. PhD thesis, University of Zurich, 2004.

[44] G. S. Iwerks, H. Samet, and K. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In *VLDB*, pages 512–523, 2003.

[45] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 25(1):157–187, 1995.

[46] A. K. Jain, M. N. Murthy, and P. J. Flynn. Data clustering: A review. Technical Report MSU-CSE-00-16, Department of Computer Science, Michigan State University, East Lansing, Michigan, August 2000.

[47] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[48] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *PODS*, pages 261–272, 1999.

[49] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing access methods for bitemporal databases. *IEEE Trans. Knowl. Data Eng.*, 10(1):1–20, 1998.

[50] D. Kwon, S. Lee, and S. Lee. Indexing the current positions of moving objects using the lazy update r-tree. In *Mobile Data Management*, pages 113–120, 2002.

[51] K. V. Laerhoven, K. A. Aidoo, and S. Lowette. Real-time analysis of data from many sensors with neural networks. In *ISWC '01: Proceedings of the 5th IEEE International Symposium on Wearable Computers*, page 115, Washington, DC, USA, 2001. IEEE Computer Society.

[52] I. Lazaridis, K. Porkaew, and S. Mehrotra. Dynamic queries over mobile objects. In *EDBT*, pages 269–286, 2002.

[53] M.-L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in r-trees: A bottom-up approach. In *VLDB*, pages 608–619, 2003.

[54] Y. Li, J. Han, and J. Yang. Clustering moving objects. In *KDD*, pages 617–622, 2004.

[55] N. Meratnia, W. Kainz, and R. de By. Spatio-temporal Methods to Reduce Data Uncertainty in Restricted Movement on a Road Network. pages 391–402. ISBN 3-540-43802-5.

[56] M. F. Mokbel, W. G. Aref, S. E. Hambrusch, and S. Prabhakar. Towards scalable location-aware services: requirements and research issues. In *GIS*, pages 110–117, 2003.

[57] M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, pages 623–634, 2004.

[58] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In *VLDB*, pages 144–155, 1994.

[59] L. O'Callaghan, A. Meyerson, R. Motwani, N. Mishra, and S. Guha. Streaming-data algorithms for high-quality clustering. In *ICDE*, pages 685–, 2002.

[60] D. Papadias, K. Mouratidis, and M. Hadjieleftheriou. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD Conference*, 2005.

[61] D. Pfoser and C. S. Jensen. Capturing the uncertainty of moving-object representations. In *SSD*, pages 111–132, 1999.

[62] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Trans. Computers*, 51(10):1124–1140, 2002.

[63] E. M. Rasmussen. Clustering algorithms. In *Information Retrieval: Data Structures & Algorithms*, pages 419–442. 1992.

[64] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, pages 1353–1356, 2004.

[65] S. Saltenis and C. S. Jensen. Indexing of moving objects for location-based services. In *ICDE*, pages 463–472, 2002.

[66] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, pages 331–342, 2000.

[67] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *ICDE*, pages 422–432, 1997.

[68] Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *SSTD*, pages 79–96, 2001.

[69] J. Sun, D. Papadias, Y. Tao, and B. Liu. Querying about the past, the present, and the future in spatio-temporal. In *ICDE*, pages 202–213, 2004.

[70] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias. Spatio-temporal aggregation using sketches. In *ICDE*, pages 214–226, 2004.

[71] Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *SIGMOD Conference*, pages 334–345, 2002.

[72] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *VLDB*, pages 287–298, 2002.

[73] Y. Tao, D. Papadias, and J. Sun. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *VLDB*, pages 790–801, 2003.

[74] Y. Tao, D. Papadias, J. Zhai, and Q. Li. Venn sampling: A novel prediction technique for moving objects. In *ICDE*, pages 680–691, 2005.

[75] N. Tatbul. Qos-driven load shedding on data streams. In *EDBT '02: Proceedings of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers*, pages 566–576, London, UK, 2002. Springer-Verlag.

[76] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.

[77] J. Tayeb, Ö. Ulusoy, and O. Wolfson. A quadtree-based dynamic attribute indexing method. *Comput. J.*, 41(3):185–200, 1998.

[78] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous queries over append-only databases. In *SIGMOD Conference*, pages 321–330, 1992.

[79] O. Wolfson. Moving objects information management: The database challenge. In *NGITS*, pages 75–89, 2002.

[80] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez. Cost and imprecision in modeling the position of moving objects. In *ICDE*, pages 588–596, 1998.

[81] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–387, 1999.

[82] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *SSDBM*, pages 111–122, 1998.

[83] O. Wolfson and H. Yin. Accuracy and resource concumption in tracking and location prediction. In *SSTD*, pages 325–343, 2003.

[84] X. Xiong, M. F. Mokbel, and W. G. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654, 2005.

[85] N. J. Yattaw. Conceptualizing space and time: a classification of geographic movement. *Cartography and Geographic Information Science*, (26):85–98, 1999.

[86] N. Ye and X. Li. A scalable, incremental learning algorithm for classification problems. *Comput. Ind. Eng.*, 43(4):677–692, 2002.

[87] B. Yu, S. H. Kim, T. Bailey, and R. Gamboa. Curve-based representation of moving object trajectories. In *IDEAS*, pages 419–425, 2004.

[88] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *SIGMOD Conference*, pages 443–454, 2003.

[89] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. In *SIGMOD Conference*, pages 103–114, 1996.

[90] B. Zheng and D. L. Lee. Semantic caching in location-dependent query processing. In *SSTD*, pages 97–116, 2001.

Appendix A

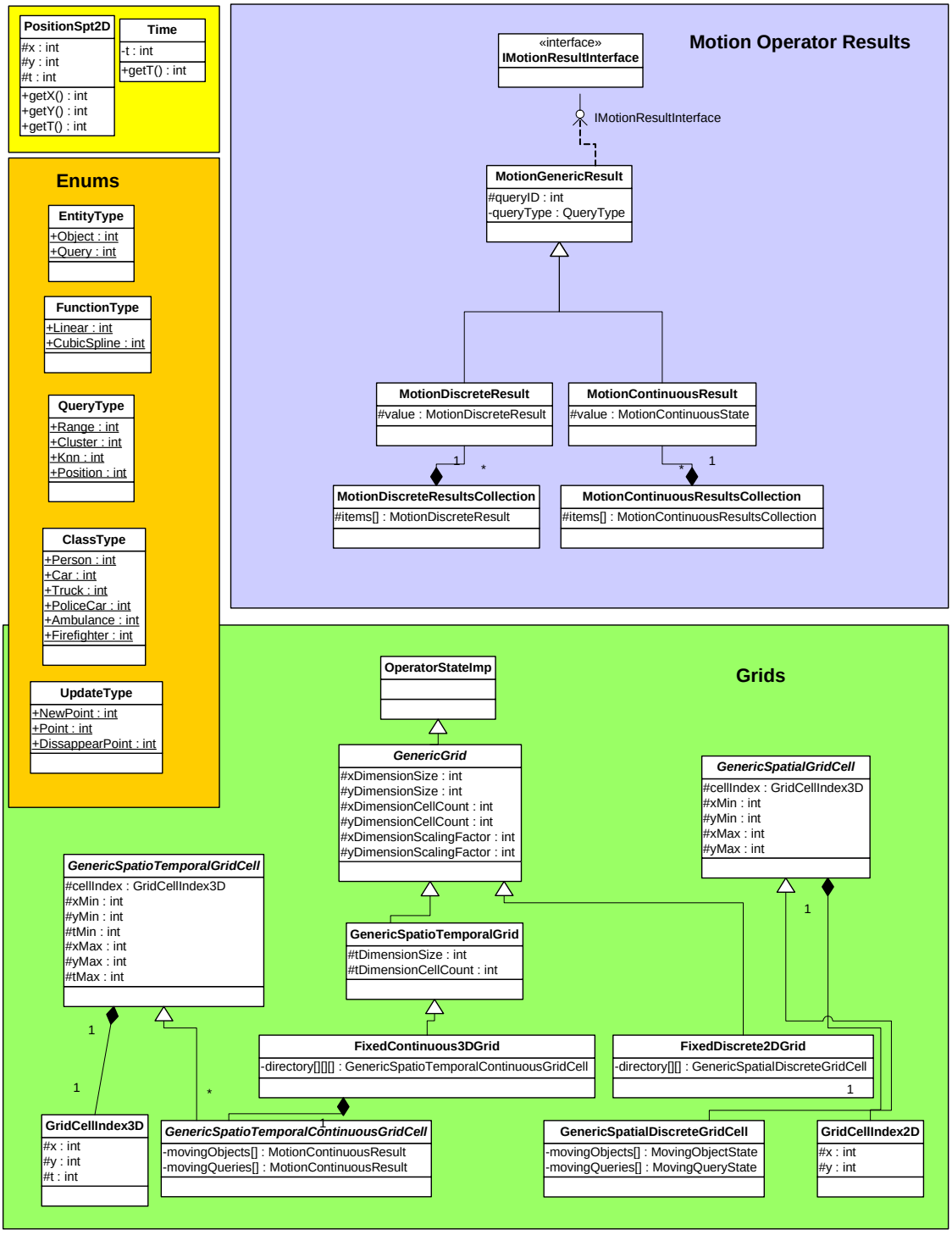


Figure A.1: UML diagram of classes used by regular grid-based motion operator in CAPE

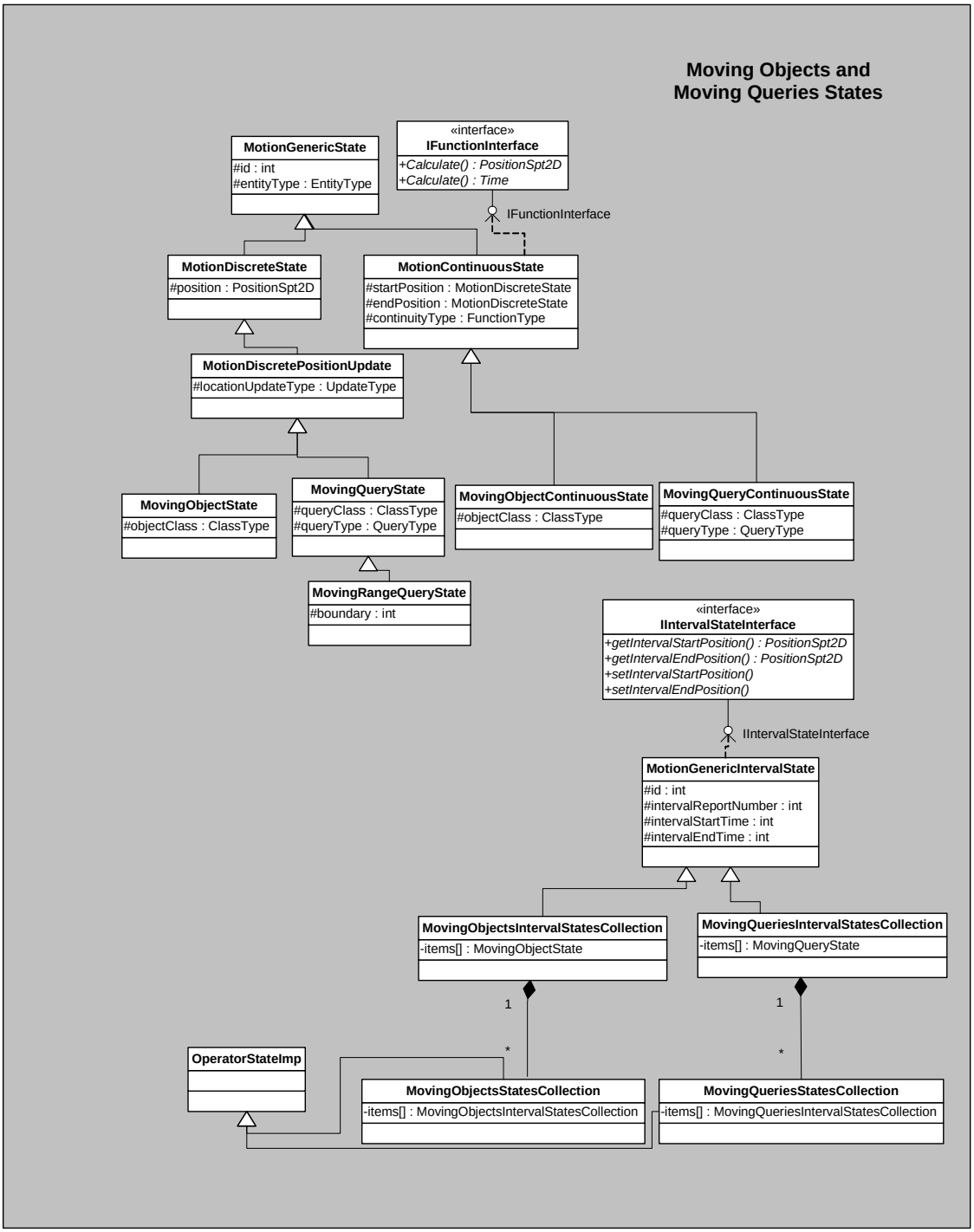


Figure A.2: UML diagram of classes used by regular grid-based motion operator in CAPE

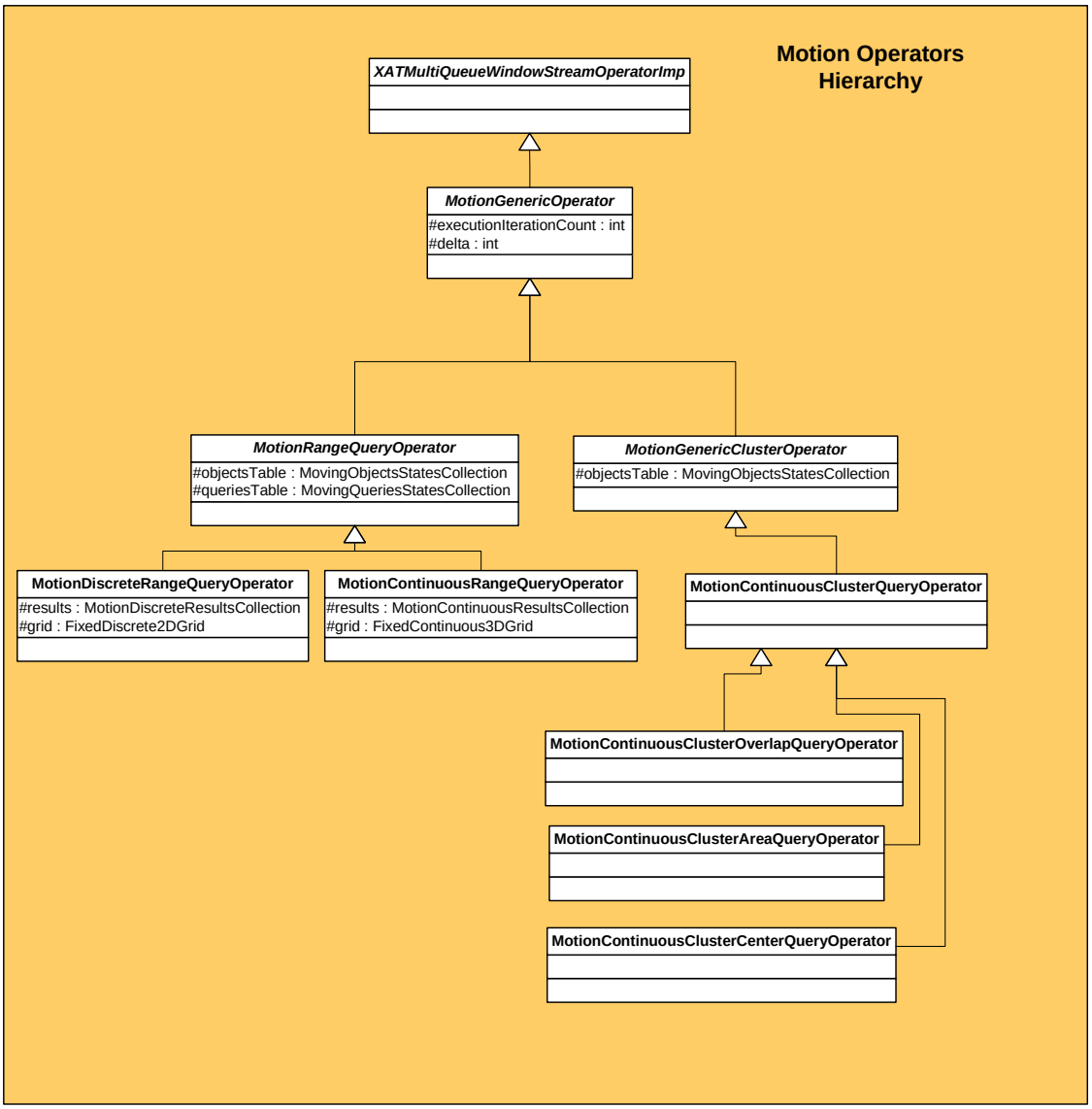


Figure A.3: UML diagram of classes used by regular grid-based motion operator in CAPE